Fine-grained vehicle classification on real traffic surveillance

A software development report submitted in partial fulfilment of
the requirements for the degree of

Master of Engineering in Software Engineering

at

Queen's University of Belfast

by

Martin O'Donnell

Table of Contents

1. System Specification

1.1 Description

This system was developed to investigate deep learning approaches to classify vehicle images in fine-grain detail. The system will train and test convolutional neural networks to investigate how various approaches perform on low resolution data. Fine-grain vehicle classification is a challenging task which is made tougher when using low resolution data. The main goal in this system is to investigate how various approaches perform on low resolution data. The main low resolution dataset used in this system will be the Boxcars dataset. This dataset provides low-resolution images that match realistic data recorded across the world.

The first approach implemented is the fine-to-coarse knowledge transfer approach. This approach takes an already state-of-the-art network, vgg16, and fine-tunes it on different resolutions of the cars-196 datasets which is a high-resolution dataset. The network is then fine-tuned again on the boxcar's dataset. This is to investigate if knowledge is transferrable between the two datasets. The next approach investigated is multitask learning. The previous method produces a single output for each image passed to the network. This approach splits up the fine-grain labels and outputs a different prediction for each part. The next approach investigated is auxiliary learning. This approach attempts to add additional regularisation to the main task, a single fine-grain label, with auxiliary tasks. In this case, a network is trained to learn a representation of a fine-grain label along with the tasks needed to make up that label. This can help regularise the network, resulting in a more generalised and robust network.

This system was designed to allow users to create various neural networks and train them with different hyperparameters without having to change the code. The system enabled this by allowing the user to add flags to the system at runtime. This will change how the networks are trained and tested. The code base is fully extendable, and new functionality can easily be added due to its design.

1.2 System Requirements

1.2.1 Dataset

- The system should be able to read in a dataset and create a dataloader to be used during training and testing
- The system should create a training, validation and testing split for each dataset
- The system should be able to read in the Boxcars dataset
- The system should be able to read in the High-Resolution Cars-196 dataset
- The system should be able to read in the Low-Resolution Cars-196 dataset
- The system should be able to make a mixed dataset from the high and low cars-196 datasets
- The system should be able to limit the number of samples from each class to create a reduced dataset (Boxcars Only)
- The system should be able to apply basic data transforms such as rotations and flips when passing data to the network
- The system should allow the user to select which dataset to use, using flags
- The user should be able to choose between single-label and multi-label versions of all datasets.

1.2.2 Convolutional Neural Network

- The system should use a state-of-the-art neural network as a base and modify the final layers to output the correct number of classes depending on the dataset used.
- The system should produce models capable of multitask learning
- The system should produce models capable of auxiliary learning

1.2.3 Training

- The system should be able to train a network to classify an image with fine-grain detail (single-label)
- The system should be able to train a multitask learning network to classify an image with fine-grain detail using multiple labels
- The system should be able to reconstruct the parts output from a multitask neural network to create a single fine-grain label
- The system should be able to train an auxiliary learning network to classify an image with fine-grain detail (single-label and multi-label)
- The system should allow the user to configure the number of epochs manually
- The system should allow the user to configure the batch size used manually
- The system should allow the user to configure the learning rate used manually
- The system should allow the user to configure the model version used manually
- The system should allow the user to configure each multitask learning loss lambdas manually
- The systems require the user to enter a model ID for each training run which must be unique
- The system should allow the user to train a network using the Adam optimiser
- The system should allow the user to train a network using the SGD optimiser
- The system should allow a model to resume training from where it left off
- The system should allow a model to be fine-tuned on a different dataset
- The system should output the training results after each epoch to a history file
- The system should save the model that receives the best validation loss during training

1.2.4 Testing

- The system should be able to evaluate a model on any dataset version
- The system should use the test split to evaluate a model
- The system should produce a confusion matrix after each test run
- The system should print the result from the tests to the console
- The system requires the user to enter the model ID to evaluate a model
- The system requires the user to enter the model version to evaluate a model
- The system requires the user to enter the dataset version to evaluate a model

1.3 Assumptions

Training a network can only realistically be completed on a high-end graphics card efficiently. The system preforms well on Tesla V100 Nvidia graphics cards taking around two to five hours to train the more complex models. By moving down to a Nvidia P100 graphics cards, the time doubled. This does work but can lead to longer processing times. The other problem is the amount of memory needed to perform an epoch. With a batch size of 32, the system will use up to 15.5GB. To decrease this, a smaller batch size can be used but this will affect the performance.

2. System Design

An overview of the system is outlined in Figure 1. The system is split up into four main sections. The datasets folder contains all the functionality to load in the datasets and split it into dataloaders to train and test a network. The model folder contains all the functionality to build the neural networks used by the system. The train_test folder includes the methods used to train and test the models. Each of these folders has an init file which the train and test scripts call to get the corresponding classes depending on arguments passed by the user. The train and test are classes where the user will interact with the system to either train or test a neural network.

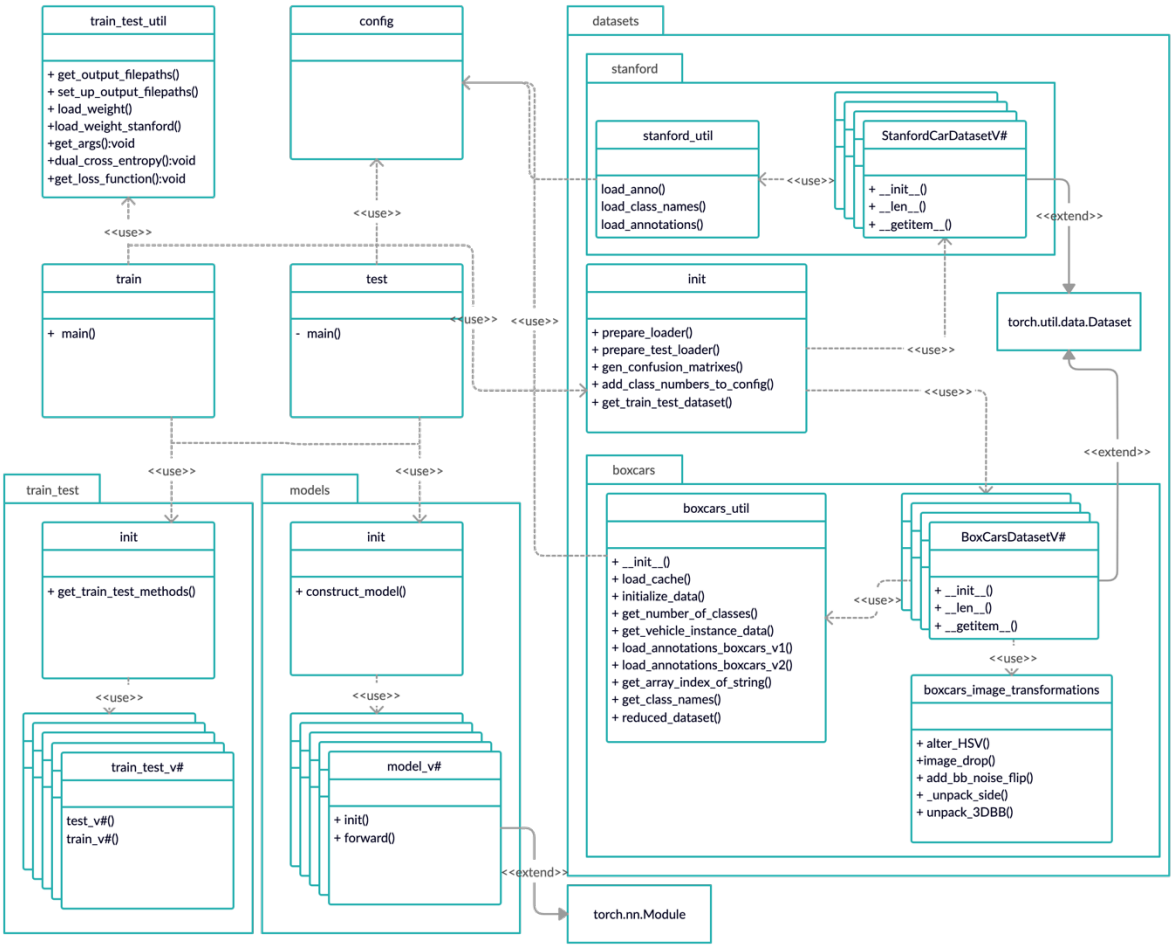

Figure 1 - UML diagram of the current system

2.1 Main Process

The user interacts with the system using the training and testing scripts. Each of these scripts links the other aspects of the system to train or test a convolutional neural network. The input that the user is able to provide to the system and their outcomes are outlined in 2.6 User Interface.

2.1.1 Training

The process of training a neural network is outlined in Figure 2. When a user runs the main script in train.py, they will be able to train a new or existing neural network. Each init file in the diagram is an interface to retrieve a particular version of that part of the system. For example, the dataset init is able to return a version of the boxcars or cars-196 dataset. The user adds specific arguments when running the training script to indicate which version to retrieve from each part of the system. Initially, the dataset, model and trainTest versions are gathered from their respective init files. A set of outputs files

are then created on the system. These will contain all the outputs during training. Finally, the loss function to use during training is decided. If the user is retraining or fine-tuning from a previous network, then the weights from this network are loaded in during the load weights step. If not, the ImageNet weights are used. The network is then trained and tested during the loop. Each loop represents one epoch. Every time the network undergoes one epoch, the results from training and testing are saved to the output files. These include the accuracy and loss of each output layer and the elapsed time. The network will use different splits of the dataset during one epoch. The training split will be used during training, and the validation is used during testing. The validation split is a small portion of the dataset to gauge the performance of the network on unseen data. The training and testing loop will repeat 32 times by default or a user-specified amount.

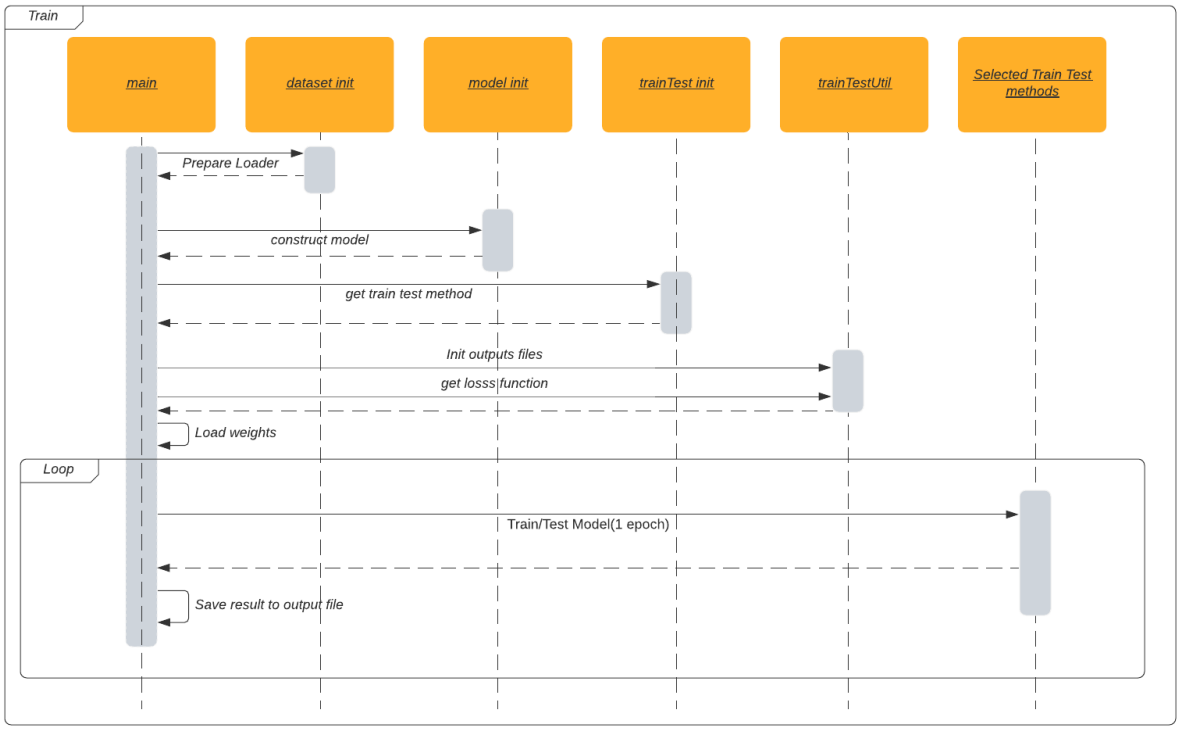

Figure 2 - Sequence diagram for the process of training a network

2.1.1 Testing

The testing script follows the same process as the training script but simpler. This is outlined in the sequence diagram in Figure 3. The user will again select the model, dataset, trainTest and loss function version as they did in training, to evaluate a network. The system will then load the weights of a network depending on the user's input. The input that is used for the testing script should be the same input used by the training script to evaluate the network on the testing split of the dataset it was trained on. During this script, the network will be tested on the test split of the dataset. This is new data that the network has not seen before. This will be used to evaluate how well the network performs on the datasets. The accuracy and elapsed time is outputted to the console when testing is finished.

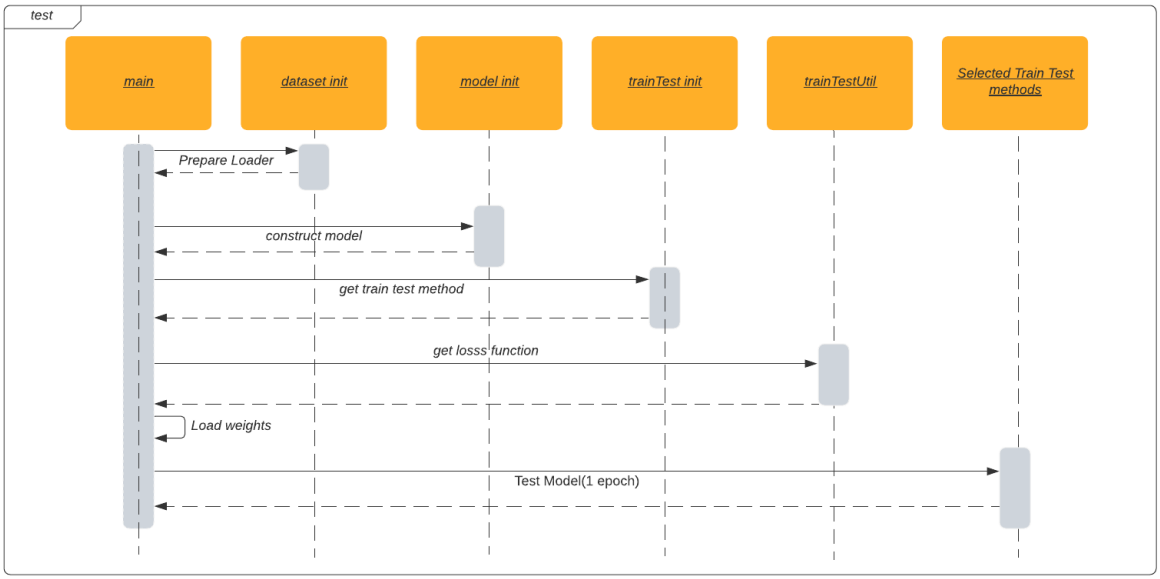

Figure 3 - Sequence diagram for the process of testing a network

2.2 Dataset

This system will allow the user to select between two datasets to train a neural network. These are the high-resolution cars-196 dataset [1] and the low-resolution boxcars dataset [2]. Both datasets are capable of training a neural network to classify vehicles in fine-grained details. The cars-196 dataset contains 16,185 images. There are 196 fine-grain labels containing information about the make, model and year of each vehicle. The boxcars dataset contains 116,286 low-resolution images. There are 107 fine-grain labels containing information about the make, model, submodel and generation of each vehicle.

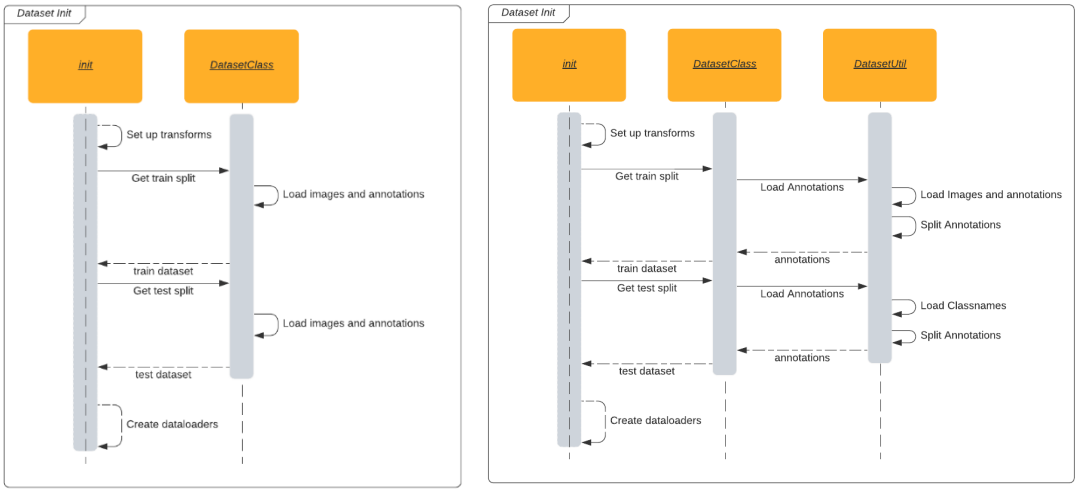

Figure 4 - Sequence diagram to initialise a dataset during training. The left one is for single-label datasets while the second is for multi-label datasets

This system enabled two versions of the cars-196 and boxcars dataset. The first is the basic version which returns a single fine-grain label for each image. The process to load this version of each dataset into the system is outlined in the left sequence diagram of Figure 4. The second version is to return the fine-grain label with the label split up into multiple parts. The process to load this version of each dataset into the system is outlined in the right sequence diagram of Figure 4. An example of the single-label and multi-labels for the boxcar's dataset can be seen in Figure 5. The diagrams in Figure

4 outline the sequence needed to retrieve the training and validation split of a dataset. The same process is used to get the testing split, but only one dataloader is returned.

In the left sequence diagram of Figure 4, the first process sets up transforms for each split. A basic set of transforms are set up to modify the images each time they are passed to the network during training. The images are transformed randomly to flip vertically and horizontal and rotate by 15 degrees to ensure the network will not fit to a specific version of the vehicle image. The validation and testing split do not receive any transforms to ensure the accuracy obtained between runs is consistent. The next step is to retrieve each split of the dataset. This system will use the PyTorch dataset class to encapsulate all the data in a split into a single class. This class will read in each image from the split, and group that with its target labels. This is done for both the training and validation split. Finally, to ensure that python can iterative over the images, they are both passed into the PyTorch dataloader object. This object will take the dataset, along with the batch size, to determine how many images are passed to the network at once.

The right diagram in Figure 4 outlines the extra steps needed to split the fine-grain label into parts. The DatasetUtil file holds all the functionality needed to do this. Each label is read in as a string. The labels are then converted to integers to allow them to be compared with the predictions from the network. To ensure that the integer representations of each label match up between each dataset split, the labels need to be read in from a file in the same order. This will ensure the ordering is consistent between splits. The fine-grain labels are then manually split up and added to the PyTorch dataset class. Each dataset is then passed into a dataloader as before, to allow python to iterate over it. When the dataset object is iterated over, an image along with the fine-grain label and parts are returned.

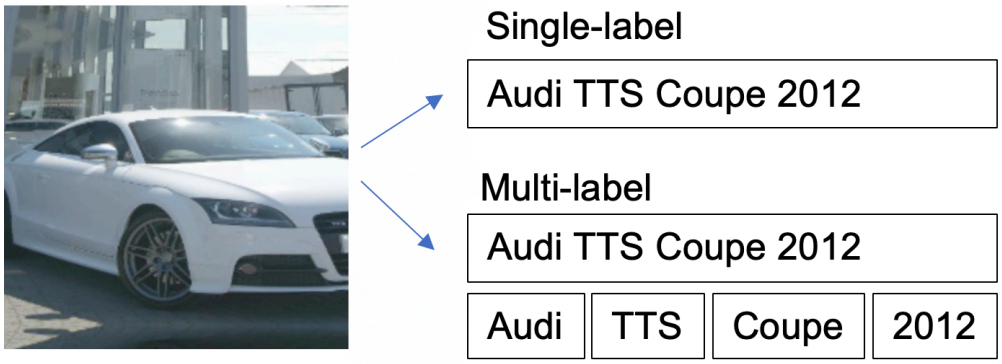

Figure 5 - Example of the single-label and multi-label dataset versions for a sample in the Boxcars dataset

Each version of the datasets has a training, validation and testing split which are fixed. The training and validation split are used during training, and the testing is used during testing. The process to return the testing is the same as Figure 4, but only one dataloader is returned.

The system also offers a number of extra dataset versions to modify the image resolution and number of samples during training to investigate the approaches outlined in 1.1 Description. Each of these datasets uses the util file to store any extra functionality needed to accomplish the desired functionality. A low-resolution cars-196 dataset needs to be created to investigate the fine-to-coarse knowledge transfer approach. A script is used to downsample the cars-196 dataset to the same resolution as the boxcar's dataset. The system is then able to decide to read in the low resolution or high-resolution images depending on the user's input. The mixed dataset containing both the low-resolution and high-resolution dataset is also created to train the network on both versions of cars-196.

To further investigate how fine-to-coarse knowledge transfer performs when transferring knowledge from the cars-196 dataset to the boxcar's dataset, a way to create a training dataset with a limited number of images is created. This version of boxcars dataset will limit the number of samples the network is shown during training. The number of samples in each class will follow the same ratio as

the whole dataset. If one class has 100 samples, and 10% of the dataset is used, then ten samples of that specific class will be used to train the network. This will only be implemented for the boxcars single label dataset. This dataset version can also return a distinct number of each class. If the user selects this option, they will be able to return a certain number of samples for each class. E.g. Return five classes from each class.

The last dataset offered uses the boxcars dataset and modified each image to follow the pre-processing proposed by paper [3]. This paper converted each image into 2D planes using a boundary box of the vehicle and passed this into the network. New data augmentation is added randomly to each image. This was done by adding random noise to remove part of the vehicle or by changing the colour of the image which will change the colour of the car.

2.3 Model

The system offers multiple neural networks with slightly different architectures to investigate each deep learning approach. Each neural network will use the vgg16 model [4]with ImageNet weights as a base. The last layer will be removed, and layers added to create the desired networks below.

2.3.1 Conventual Neural Network

The first type of neural network will output a single label for each input. This is done by adding a single output layer to the base network. There are 196 outputs when the cars-196 dataset is used and 107 outputs when the boxcars dataset is used.

2.3.2 Multitask Learning Neural Network

The second type of neural network is created to accomplish multitask learning. These networks will add an output layer for each part from the fine-grained datasets. For the cars-196 dataset, three output layers are added to provide predictions for the make, model and year of a vehicle. For the boxcar's dataset, four output layers are added to provide predictions for the make, model, submodel and generation of a vehicle. A number of different final layer configurations were added to the system to investigate how passing data between the final layers affects performance. Figure 6 describes the architecture of these networks. Model A passes the output from the fc2 layer into each output layer. Model B passes the output from fc2 layer into each output layer. The prediction from the higher-level layers is passed into the lower level layers to give each layer more information to base their predictions on. Model C passes the output from fc2 into the make. The prediction from each higher level output is passed into the lower level output layers. These three networks were created to investigate if modelling the final layers in a hierarchical format would affect the performance.

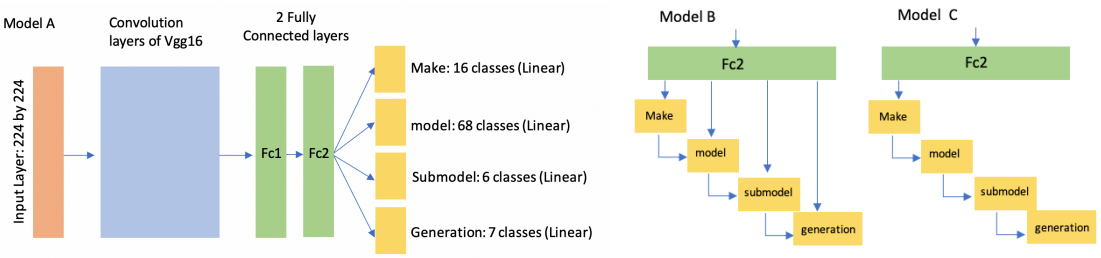

Figure 6 – Multitask learning networks with Vgg16 as base and boxcars output layers. Model B and C are simplified diagrams outlining different versions of data passing between the final output layers. Every yellow box in these diagrams will give an output when an input is passed to the model. Each fully connected layer consists of a linear layer with input and output of 4096 features followed by a dropout layer with a probability of 0.5. The output layers are all linear layers with the number of output equal to the number of unique labels for that task

2.3.3 Auxiliary Learning Neural Network

The third type of network is created to accomplish auxiliary learning when the boxcars dataset is used. This neural network will combine the output layers in 2.3.1 Conventual Neural Network and 2.3.2 Multitask Learning Neural Networkto create a model that outputs both the full and separated

fine-grain labels. These networks will only work for the boxcar's dataset. An example of the architectures is outlined in Figure 7. The process of passing data between the auxiliary tasks follows that of the final layers of model A in Figure 6. Auxiliary networks are also created to model the data flow of the final layers in model B in Figure 6. Model D outlines the first auxiliary network created in the system. This network provides a prediction for each auxiliary, along with the main task. These predictions are then passed into the final output layer to re-predict the full fine-grain label. Model E calculates the prediction from each auxiliary task and passes these to the final layer along with the output from the fc2 layer. Model F passed the prediction of the auxiliary tasks to the output layer.

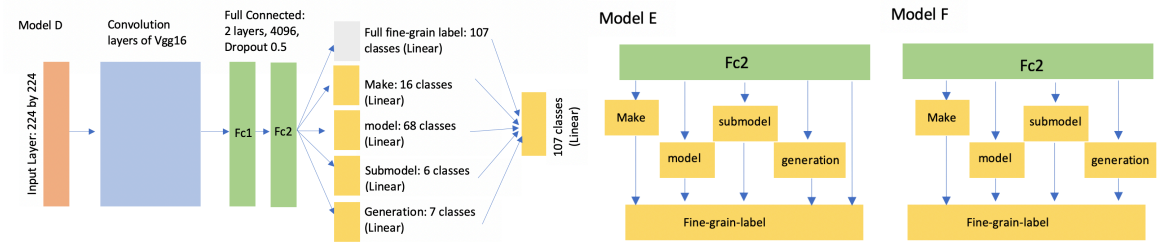

Figure 7 – Auxiliary learning models D, E and F. They outline different ways to pass information to the main task. The yellow boxes indicate which layer returns an output when an image is input into the network

2.3.4 Channel Max Pooling

This neural network takes the model from 2.3.1 Conventual Neural Network and adds a channel max-pooling (CMP) layer between the last max-pooling layer and the first fully connected layer. This new layer, proposed in [5], showed that there was a performance boost when used on high-resolution datasets. This additional layer will modify the feature map passed into the fully connected section of the neural network. The feature map will change dimensions from 512 x 7 x 7 to c x 7 x 7. c is calculated by taking the number of channels from the last max-pooling layer (512) and dividing it by the compression factor r. When using a compression factor of two and the vgg16 model, the first layer in the fully connected section is modified to contain 12,544 in features.

2.4 TrainTest

The trainTest section contains the methods to train and test a neural network. One trainTest version includes two functions, one to train the network and another to evaluate it. These methods are mostly the same, but the creation of confusion matrixes is removed in the training methods to increase performance. Creating a generalised trainTest method to work on all model and dataset configurations causes the training time to increase. A separate trainTest version will be generated for each model and dataset configuration to speed up training times. This approach will cause a lot of code to be duplicated but dramatically improves performance during training. Each trainTest version completes one epoch of training or testing. The sequence for training a network is outlined in Figure 8.

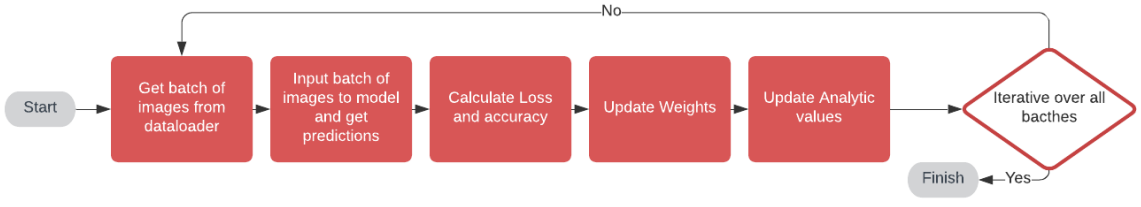

Figure 8 - Sequence diagram to train a model.

Dataloaders are used to iterative over the dataset in batches. Each iteration is called a training step. A batch of images are retrieved from the dataloader and passed into the model, and a prediction for each is returned. The loss and accuracy are then calculated from the prediction, and the weights of the model are updated. These steps are repeated until all the images in the dataset have been passed through the model. The testing method follows the same process as Figure 8 without updating the

weights of the model. The analytic values that are updated during training and testing are the accuracy, loss and elapsed times. These are saved to the output file after each epoch.

The same test method is used when the user interacts with the training and testing scripts. A confusion matrix will only be created when the user interacts with the testing script because the process is time-consuming and can lead to a high number of output files if there are a lot of epochs. During the testing phase, a confusion matrix is made for each output layer. This gives an overview of how each image in the testing split was predicted. These files are saved in the saves/confusionMatrix folder.

2.5 Loss Function

The cross-entropy loss function is the primary loss function applied during training. This is the default loss function used when training occurs. The user is able to swap out loss functions by passing arguments to the system. A second loss function is added to validate if dual cross-entropy loss proposed in [6] affects the performance on low-resolution images. This loss function is currently not working but will be fixed in future iterations of the system.

2.6 User Interface

This system will be run using the command line. It is expected that a user working with this system will have experience using the command line. The package argparse is used to allow the user to pass arguments through the command line to the python scripts.

2.6.1 Training and Testing

All aspects of training a neural network are fully configurable. This results in a large number of arguments that the system needs to handle. Information about the required arguments and their flags are found in Table 1. The required flags are explained further in 2.6.2 Dataset, 2.6.3 Models and

2.6.4 TrainTest. The hyperparameters values can affect training significantly, so a way to change these easily has been added to the system. These flags are outlined in Table 2. All flags except the hyperparameters specifically used to train the network can be used when testing the network.

Flag	Default	Description
--model-id	N/A	A unique identifier that is added to all outputs during each training run. This is manually inputted by the user. This must be unique between runs.
--dataset-version	1	Dataset used to train a neural network
--train-test-version	1	Training and Test method version used to train the network
--model-version	1	Model architecture used
--adam	False	This flag uses the Adam optimiser over SGD. This optimiser creates networks that perform better in every instance on the system. Add the flag as a parameter to set this flag to True

Table 1 - Required flags needed to train or test a neural network

Flag	Default	Additional Notes
--batch-size	32	Batch size
--epochs	40	Number of epochs
--lr	1e-4	Learning Rate
--weight-decay	0.0001	Weight Decay for SGD. Only used when the Adam flag is false
--momentum	0.9	Momentum Decay for SGD. Only used when the Adam flag is false

--train-samples*	1	Can only be used when the boxcars dataset is selected. Will decrease the dataset to contain 'x' number of samples for each class
--train-sample-percent*	False	Added to convert the train-samples value to a percentage of the samples from each class in boxcars
--ds-stanford*	False	Added to use the low-resolution cars-196 images. Requires the dataset-version flag to be set to 1
--main-loss	1	Lambda to multiply the main task in Equation 2 to train auxiliary learning networks
--make-loss	0.2	Lambda to multiply the make task in Equation 1 and Equation 2 to train multitask and auxiliary learning networks
--model-loss	0.2	Lambda to multiply the model task in Equation 1 and Equation 2 to train multitask and auxiliary learning networks
--submodel-loss	0.2	Lambda to multiply the submodel task in Equation 1 and Equation 2 to train multitask and auxiliary learning networks
--generation_loss	0.2	Lambda to multiply the generation task in Equation 1 and Equation 2 to train multitask and auxiliary learning networks
--testing	False	When true, the system will save no outputs.

Table 2- Flags used to change the hyperparameter when training a network

* Explained further in 2.6.2 Dataset

The system offers the functionality to continue training a model or fine-tuning it from an existing model. The flags used during fine-tuning are outlined in Table 3. The fine-tune argument must be added to alert the system that this functionality will be utilised. To continue training where the network left off, the user will use the same parameters they used to train the initial model but with a new model ID. This must be unique with every run. The user will then add the fine-tune-id equal to the model ID they used to train the previous model. The commands needed to do that are outlined in Figure 9.

Initiate Training *python3 train.py –model-id 33 –model-version 2 –dataset-version 2 –adam*

Continue Training *python3 train.py –model-id **34** –model-version 2 –dataset-version 2 –adam –finetune –fine-tune-id 33*

Figure 9 – The top command is used to start training a network using the boxcars datasets. The second command is used to continuing the training the network from where it left off.

Flag	Default	Additional Notes
--finetune	False	Enables the functionality of fine-tuning to occur
--fine-tune-id	N/A	The weights are taken from this model and used for the training run
--fine-tune-stan-box	False	Allows training a model using the weights of a previous model with different output layers

Table 3 - Flags used to fine-tune a network

Fine-tuning a network on a different dataset involves changing the architecture of a network to ensure the final layers have the correct number of classes. The fine-tune-stan-box flag allows this to occur. There are limitations to this feature. This will only work when the fine-tuned model base is trained on the single-label cars-196 dataset. The current system is not set up to go from a boxcars model to cars-196 model.

There are a number of flags that are available to the user but will cause the system to error if changed. (Table 4) These flags are used by the current system with their default values. They were added to test features at the start, which were not implemented in the final system. They were kept in because they will be added in future iterations of the system

Flag	Default	Additional Notes
--imgsize	224	The size images are normalised before input into each neural network
--boxcar-split **	Hard	Boxcars dataset has multiple splits. The system will only work with the hard split
--loss-function***	1	Selection of loss function.

Table 4 - Flags that are available to the user but they are currently not able to change from their default values

** Explained further in 3.1.1 Datasets
*** Explained further in 2.6.5 Loss Function

2.6.2 Dataset

A user will use the dataset-version and train-test-version flag to select between the cars-196 and boxcars dataset. The system provides multiple variations of both datasets. The configurations of the flag to receive a specific dataset are explained in Table 5. To use the low-resolution images, the user will add the ds-stanford flag when running the script. This will work on both labelling versions of cars-196. The dataset version must be set to one to use this feature. The train test flag is used here to ensure the dataset labelling version links up to the trainTest methods outlined in

2.6.4 TrainTest.

Dataset Version	Train Test Version	Output
1	1	Cars-196 dataset (single-label)
1	2	Cars-196 dataset (multi-label)
2	1,7	Boxcars dataset (single-label)
2	Not 1 or 7	Boxcars dataset (multi-label)
3	1 Only	Boxcars dataset (single-label) with data augmentation from [3]
4	1 Only	Boxcars dataset (single-label) with a limit on the number of samples for each class
5	1 Only	Mixed Cars-196 dataset (combined low and high-resolution dataset) (single-label)

Table 5 - Dataset configuration offered by the system

Dataset version three and four are special cases of the boxcar's dataset. These versions will only return a single-label for each sample. When version three is used, the images in the dataset will be converted into a 2D plane using the boundary boxes. This is to mimic the input seen in the Boxcars paper.[3] When the dataset version is set to four, the dataset will use an additional flag to determine how many samples will be processed for each class. The train-sample flag will be used in this instance. If the train-sample-percent file is used, a percent of samples for each will be used instead.

Dataset version 5 is used to create the mixed cars-196 dataset with both high and low-resolution images. The high-resolution dataset is downsampled to the resolution of 122 x 122 to form the low-resolution dataset. This is completed using an external script. The images that are downsampled contain _ds in their filename. Both these datasets are combined to create a mixed cars-196 dataset. Dataset version 3 and 5 are used to test knowledge transfer between the cars-196 dataset and boxcars, while version 4 is used to validate the results seen in the paper [2].

2.6.3 Models

The user is required to select which model to use by setting the model-version flag to a value seen in the table in Table 6, Table 7 or Table 8. Each network will use the vgg16 architecture as a base and the pre-trained ImageNet weights unless otherwise stated. The networks can be split into two groups. Networks that input and output a single-label will be called single-label models while the networks that input and output multiple labels will be called multi-label models. The design of each network architecture is outlined in 2.3 Model.

2.6.3.1 Single Label Neural Networks

Model Version	Description
1	The base model's last layer is modified to output the number of classes in the dataset used. The fully connected layers are initialised with random weights
7	The base model's last layer is modified to output the number of classes in the dataset.
8	The base model with the last layer modified to output the number of classes in a dataset. CMP layer is added between the last max pooling and first fully connected layer
14	AlexNet model with the last layer modified to output the number of classes in a dataset. Weights are initialised from ImageNet

Table 6 – Single label neural networks configuration

2.6.3.2 Multilabel Multitask Learning Neural Networks

Model Version	Description
2,10,11	Multitask learning neural networks for boxcars. The different architectures are outlined in Figure 6(A-2, B-10, C-11)
3	Multitask learning neural networks following the architecture of Model A (Figure 6) Each output layer has its own fully connected layer (green boxes)
4	Multitask learning similar to model A(Figure 6), but a dropout layer with a probability of 0.2 is added before the final output of each task
21	Multitask learning network for cars-196. Follows the same format as Model A(Figure 6)

Table 7 - Multitask learning neural Networks

2.6.3.3 Multi-label Auxiliary Learning Neural Networks

Model Version	Description
5,12,13	Auxiliary Learning neural network. The difference in architectures are outlined in Figure 7 (5-E,12-D,13-F). The fully connected layer uses random weights instead of ImageNet weights
6	Auxiliary Learning neural network. Follows the same architecture as Model D (Figure 7). Each task has its own fully connected layers (green boxes)
15,16,17	Auxiliary Learning neural network. The difference in architectures is outlined in Figure 7 (15-D,16-E,17-F). The auxiliary tasks follow the data flow from model A
18,19,20	Auxiliary Learning neural network. Follow the same architecture as Figure 7, but the auxiliary tasks follow the data flow from model B

Table 8 - Auxiliary learning neural networks

2.6.4 TrainTest

The train-test-version flag is used by the user to determine which method to use during training and testing. There are a number of different multi-label trainTest versions implemented to ensure the correct loss functions is used to train the model. The available configurations can be seen in Table 9.

TrainTest Version	Model	Dataset
1	Single-label	Single-label and Multi-label
3	Multi-Label – Multitask learning	Cars-196 Multi-label
5	Multi-Label – Multitask learning	Boxcars Multi-label
6	Multi-Label – Auxiliary learning	Boxcars Multi-label
7	Multi-Label – Multitask learning	Boxcars Multi-label

Table 9 - trainTest configuration use by the system

The trainTest version 7 will only work when the network is evaluated using the testing script. It is used to evaluate how well a boxcars multitask neural network performs in calculating the full fine-grain label. This version will be reconstructing the fine-grain label from the separate predictions and comparing this against the full label to calculate the accuracy.

2.6.5 Loss Function

The system enables the user to select between two loss functions to calculate the loss from a single task. These are seen in Table 10. The current system only works with the categorical cross-entropy loss function. The dual cross-entropy function has not been implemented correctly yet. This will be added in future improvements.

Loss function value	Loss Function used
1	Categorical Cross-Entropy
2	Dual Cross-Entropy [6]

Table 10 - Loss function offered by the system.

2.6.6 Configuration File

The config file is used by the system to locate where files, that are used by the system, are saved. The output files are saved in the save folder. The datasets are stored in the data folder. The user is able to change these locations by changing the file path in this file. This file can be easily modified by the users when the system is installed without having to delve into the code base.

3 Implementation

3.1 Languages, packages, and libraries

The development language was implemented in python. The main library used was pytorch[7]. It is an open-source machine learning library. This library provides functionality to train and test neural networks. The torchvision package was used to load in the vgg16 architecture and ImageNet weights. Pandas and matplotlib were used to visualise the results output during training. To train the neural network with images, they need to be in a specific format. The PIL package was used to do this. The system allows the user to change how the networks are trained using arguments. The argparse package manages this and provides the user with the ability to add inputs to the system from the command line.

3.1.1 Datasets

The datasets information is read into a class that extends from the abstract Dataset class from PyTorch. Each dataset class uses the init, len and next method to define the dataset. The init file

collects the filenames, and target labels of the dataset split into a dictionary. When the next method is called, a samples filename is retrieved, and the image formatted using PIL is obtained. This is then returned along with the target labels. The len method is used to return the number of samples contained in the dataset. The newly created dataset object is then passed to a dataloader class to allow python to iterate over it. This is used during training and testing to show the model each sample from the dataset. Training a network one sample at a time is not efficient or optimum for training a network, so it is passed into a dataloader to allow the network to receive batches of samples at once.

To convert from the single-label to multi-label datasets, there is extra processing completed in the init method. For the boxcar's dataset, the single label is split by the spaces to retrieve four parts of the label. These are the make, model, submodel and generation. The cars-196 dataset goes through a similar process. An extra step is needed for the cars-196 dataset because some make, and model values have multiple words, so they need to be concatenated. The code to complete the cars-196 split up process was taken from the Car-Model-Classification git repo.[8] The full label and split up parts are all returned when the dataloader retrieves a sample from the dataset.

To downsample the cars-196 dataset, an external script is used to create a low-resolution version of each image. When the user installs the system and downloads the cars-196 dataset from the authors, the low-resolution data will need to be generated. To do this, the user will run the DownSampleImages script in the jupyter_files folder. The user will need to update the path to the root folder of the dataset, if not placed in the data folder. The low-resolution images will have a _ds appended to the end of each filename to indicate that it is the low-resolution version.

3.1.2 Models

Each model in the system extends from the nn.Module class in PyTorch. This is the base class for all neural network modules. Classes that extend this can overwrite the init and forward methods. The init method is used to create the architecture of the neural network. The forward method is used to transfer data through the network to calculate the final outcomes. Each network implements its own version of these methods to create their custom architecture and data flow. The network architectures are explained in 2.3 Model. To adjust how data is passed through the network, the forward method is updated. The final layers dimensions need to be updated to match that of the forward method to ensure the input to each layer matches the feature map from the previous layer. This is why there are so many networks in this system.

3.1.3 Loss Functions

The process of training a neural network is outlined in 2.4 TrainTest. One principal factor in training is calculating a loss that will update the weights to train the network. All approaches offered by this system need to take into account all output layers to calculate an overall loss. For each network, the output layers will calculate their own loss using the categorical cross-entropy loss function. This is used to compare the distributions of the predictions with the true distributions. For models that have a single output layer, this loss function is used. For multitask and auxiliary learning networks, multiple tasks are output and these need to be taken into account to calculate an overall loss. Each output layer will calculate their own loss using the categorical cross-entropy loss function. Equation 1 and Equation 2 are used to calculate the total loss for the boxcar's dataset. Each individual loss is multiplied by a separate lambda to affect how much that task weighs on the total loss. This can be used to model different relationships between the tasks.

$$loss_{boxcar} = loss_{make} * \lambda_{make} + loss_{model} * \lambda_{model} + loss_{submodel} * \lambda_{submodel} + loss_{generation} * \lambda_{generation}$$

Equation 1 – Loss functions used when multitask learning networks are trained on the boxcar's dataset.

$$loss_{boxcar} = loss_{main} * \lambda_{main} + loss_{make} * \lambda_{make} + loss_{model} * \lambda_{model} + loss_{submodel} * \lambda_{submodel} + loss_{generation} * \lambda_{generation}$$

Equation 2 – Loss functions used when auxiliary learning networks are trained on the boxcar's dataset

3.2 Development Process

At the start of the project, a substantial amount of reading was completed to learn more about the topic and plan what the system would be able to do. Once the main topics were decided, development began. The main ideas were there, but there was no way of knowing how the techniques would perform because they have not been done on low-resolution data. This project followed a similar path to agile development. An idea was derived from the research and the path I believed the project should take. This was then designed, developed and tested. The results were evaluated to understand how the new idea performed and was used to decide what the next step forward would be. This was used significantly during the multitask and auxiliary learning section. This idea initially started off with one multitask neural network along with the loss function outlined in Equation 1. A model was trained and evaluated. The performance was good, but there was no way of comparing it to anything else because each output gives its own performance. The idea of reconstructing the full fine-grain label came about so that the performance could be evaluated with the baseline. When this was done, a worse performance was obtained but sparked more ideas on how different architectures and loss function configurations would affect the result. This led to modelling the relationships in a hierarchical format similar to the fine-grain labels. Once this was implemented the best results were found. All the research from this section was used as the foundation of the auxiliary learning section, which uses the multitask learning architectures to build the auxiliary tasks for the auxiliary networks. This iterative process was completed for each section of the project to ensure the best outcome and every possible avenue was explored.

3.3 Environment

Training a neural network consists of computing millions of mathematical operations per run. This can lead to very high processing times. A graphic processing unit (GPU) is best suited for this task. For the majority of the project's lifecycle, the training and testing were completed on the Kelvin server in Queens. This server contains 4 Tesla V100 Nvidia graphics cards. In the last month of the project, a fault occurred in Kelvin, so google cloud servers [9] was used to finish the final testing. The same graphic card was used to ensure that the results remained consistent.

All communication between myself and the servers was conducted on the command line using SSH. Training a neural network can take a long time, so slurm [10] was used on kelvin to run the scripts in the background over many hours. Nohup [11] was used on google cloud servers to do the same task. It was found that running the scripts over multiple GPUs gave a worse result, so it was decided to run each model on its own dedicated graphics card. This allowed multiple models to be trained at the same time whilst using Kelvin.

All the analysis of the training and testing data was completed on my own computer, so the output files needed to be transferred off the servers. Transferring files can take a long time due to their size, so FileZilla [12] was used. This program provides a graphic interface showing the file structure of my own computer and the server. This sped up the time taken to find the right file and send it over.

The system was saved on my personal GitHub during the span of the project. This ensured that the system was always backed up and a log of everything was kept. It also made it very easy to move the code between servers. Git was installed on all the servers, and I just needed three commands to get the system onto the server. The datasets are large and were not stored on GitHub. I saved them in a google cloud bucket [9] to allow any server to download them on the command line.

4. Testing

The pytest package was used to test the system. This is a lightweight package that offers multiple features to test all aspects of the system fully. Sixty-Nine tests are written to test the whole system. The system has broken up into subfolders and classes to encapsulate the functionality. The testing files followed this format. Testing a machine learning project can be difficult due to the complexity. Every time a neural network is trained, it will output different values due to the seeds and the weights of a model. This was taken into account when testing the system. All tests were tested using the assert function from pytest. This is used to compare two conditions. If an error occurs, the test will fail. To run the test files, cd into the unit_test folder and run the pytest command.

To test the model classes, a single image is passed into the model, and a set of predictions are returned. The loss is calculated, and the weights were updated in the model. The weights before and after the update are compared to ensure that each layer in the network trained correctly. The model should output values between certain ranges. These can also be tested for. The number of classes in each prediction is tested to ensure they match the boxcar and cars-196 datasets.

Testing the datasets is completed by manually verifying the number of samples in each dataset. The system only allows the boxcars and cars-196 dataset to be read in so the number of each sample is known. To test the dataset classes, each split is read in, and the number of samples are compared against the known values. The data type and size of the images are also checked to ensure that they are all in the correct format. This section also has a number of different methods located in the utility files that are used to pre-process the data before it is added to the dataset. Each of these methods are tested separately.

Testing the trainTest methods follows the same routine as the models. A model is created and is trained using the training method. The weights before and after the training method are compared to ensure they are updated. A mocked dataloader is used to only pass in a single image. This speeds up testing times. The training method returns parameters on how the training step performed. This is checked to ensure they have been updated and are not equal to zero. To test the testing methods, the same process occurs, and the return values are checked, so they are not set to zero.

There are various helper files used by the system to complete separate functions like setting up a file with a particular name for the outputs or return the loss function used by the training methods. These are all tested separately to ensure they do what they are meant to do.

To integrate each aspect of the system together, the training and testing script are tested. A single test is written for both of them to ensure that all aspects of the system are correctly connected. These tests may take some time because a full network is trained over a single epoch. The smallest dataset, boxcar limited, is used to train the network to speed up these tests.

5. Conclusion

The system has been developed to enable researchers to train and evaluate neural networks on low-resolution datasets. The system outlined has been developed in a way that is fully extendable so new features can easily be added. There are also a wide variety of arguments that can be passed to the system already that ensure that every aspect of training and testing a network can be done without having to understand the code. The four approaches, fine-to-coarse knowledge transfer, multitask learning, auxiliary learning and a channel max-pooling network, have all been implemented in this system. The system has allowed a thorough investigation of each approach on low-resolution data. Future additions to the system would be improving the performance of the CMP layer, implementing dual cross-entropy and setting up auxiliary learning for the cars-196 dataset.

6. References

[1] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3D object representations for fine-grained categorization," in *Proceedings of the IEEE International Conference on Computer Vision*, 2013, pp. 554–561.

[2] J. Sochor, J. Špaňhel, and A. Herout, "BoxCars: Improving Fine-Grained Recognition of Vehicles Using 3-D Bounding Boxes in Traffic Surveillance," *IEEE Trans. Intell. Transp. Syst.*, vol. 20, no. 1, pp. 97–108, 2019.

[3] J. Sochor, A. Herout, and J. Havel, "BoxCars: 3D Boxes as CNN Input for Improved Fine-Grained Vehicle Recognition," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016, vol. 2016-Decem, pp. 3006–3015.

[4] S. Liu and W. Deng, "Very deep convolutional neural network based image classification using small training sample size," in *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, 2015, pp. 730–734.

[5] Z. Ma *et al.*, "Fine-Grained Vehicle Classification with Channel Max Pooling Modified CNNs," *IEEE Trans. Veh. Technol.*, vol. 68, no. 4, pp. 3224–3233, 2019.

[6] X. Li, L. Yu, D. Chang, Z. Ma, and J. Cao, "Dual Cross-Entropy Loss for Small-Sample Fine-Grained Vehicle Classification," *IEEE Trans. Veh. Technol.*, vol. 68, no. 5, pp. 4204–4212, 2019.

[7] "Pytorch." [Online]. Available: https://pytorch.org/.

[8] Kamwoh, "Car-Model-Classification." [Online]. Available: https://github.com/kamwoh/Car-Model-Classification.

[9] "Google Cloud." [Online]. Available: https://cloud.google.com/.

[10] "Slurm." [Online]. Available: https://slurm.schedmd.com/documentation.html.

[11] "Nohup."

[12] "Filezilla." [Online]. Available: https://filezilla-project.org/.