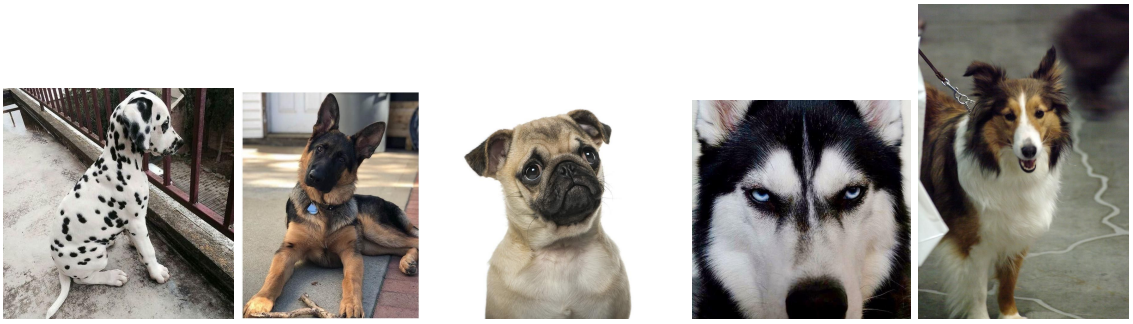
機器學習期末專案報告

成員：戴宏周 0416302、顏靖剛 0416092、呂承祐 0416010

一、主題

狗種圖片辨識(dog image classification)。針對以下六種常見狗種圖片進行識別：

1. 大麥町(Dalmatian)
2. 德國牧羊犬(German Shepherd)
3. 巴哥犬(Pug)
4. 哈士奇(Husky)
5. 喜樂蒂(Sheltie)
6. 柴犬(Shiba)



以上圖片依序為：大麥町、德國牧羊犬、巴哥犬、哈士奇、喜樂蒂、柴犬。

二、作法

A. 資料蒐集

資料來源為使用 google-image-downloads(https://github.com/hardikvasa/google-images-download)爬蟲將Google Search中圖片結果爬下來。圖片搜尋設定如下：

- medium size(約300 pixel width ~ 600 pixel width)
- 800張，但有部份無法爬取
- 以jpg副檔名爬取，然而實際上這選項並無效用

B. 資料前處理

將各個class中不適當圖片剃除，剃除主要依照以下條件：

1. 圖片中含有其他狗種
2. 圖片中沒有狗
3. 圖片非照片(手繪、電腦繪圖、動畫)

最後保留3708張合法圖片、以及723張非法圖片。
將所有圖片進行轉檔為jpg，並進行resize。resize成以下幾種：

1. 224X224
2. 64X64
3. 32X32
4. 短邊為256，長邊依照比例縮放

將最終資料3708張合法圖片分為：train dataset、validation dataset、test dataset，比例為70%、10%、20%。
資料進行mean substraction正規化，並且進行data augmentation(ex: random crop, random flip)來擴大資料集。
其他訓練方式使用PCA以及Autoencoder進行特徵抽取作為訓練前處理。PCA使用GridSearch，進行有限度暴搜，而Autoencoder使用兩種方式：

1. Multilayer perceptron
2. Convolution Neural Network

並將結果匯出為tsv檔。

C. PCA with SVM

先透過PCA對train set與test set進行feature抽取，主要使用224X224之圖片，並依照以下設定進行抽取，分別為：

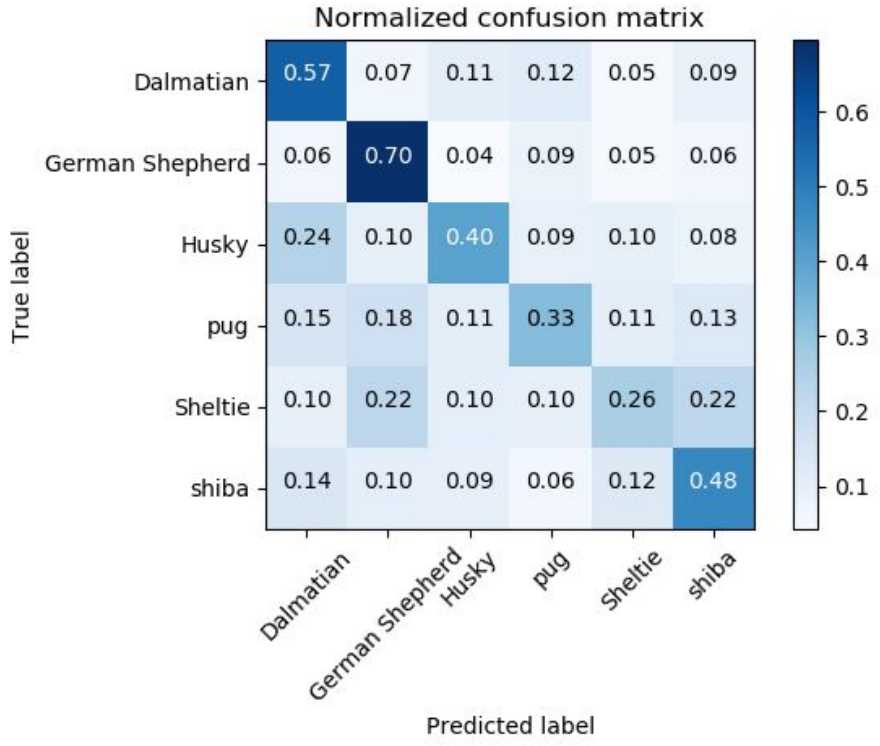
1. Component 50 with RGB
2. Component 100 with RGB
3. Component 150 with RGB
4. Component 50 with grayscale
5. Component 100 with grayscale
6. Component 150 with grayscale

抽取完之結果透過SVM進行分類(使用GridSearchCV)進行SVM fitting(train dataset)。最後將test dataset經過PCA後結果放入SVM進行分類。最後獲取準確度與Confusion matrix。

	precision	recall	f1-score
Component 50 RGB	0.45	0.46	0.45
Component 100 RGB	0.42	0.42	0.42
Component 150 RGB	0.43	0.43	0.43
Component 50	0.34	0.34	0.34

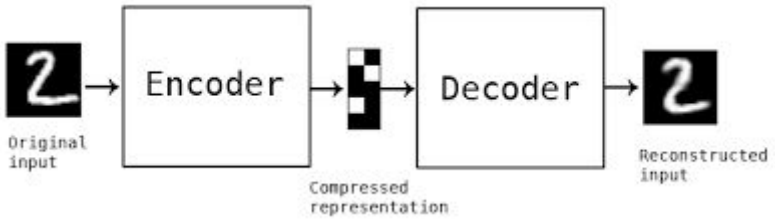
grayscale			
Component 100 grayscale	0.31	0.31	0.31
Component 150 grayscale	0.31	0.31	0.31

下圖為此法最佳解Component 50 RGB之Confusion matrix。

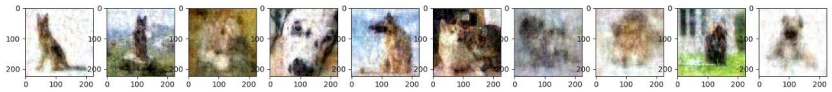


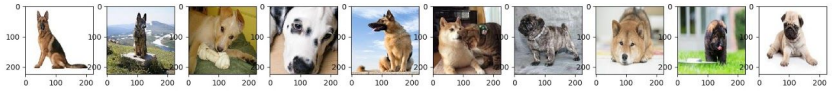
D. Autoencoder with SVM

Autoencoder 是一種降維策略，可以透過Multilayer perceptron或是CNN進行降維，此方法需要進行訓練，目標是能盡量回覆原始資訊。以下我們透過Multilayer perceptron或是Convolution Neural Networks進行降維。最後透過SVM進行分類。所有資料均使用224X224圖片大小。但礙於時間的關係，我們並沒有直接將autoencoder和svm相連後進行訓練，所以在NN的部份其實是沒有受到label影響，故結果和PCA差不多。

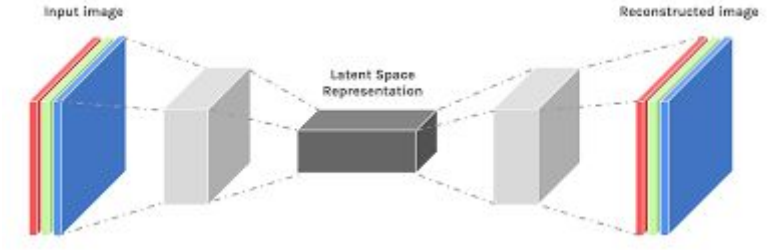


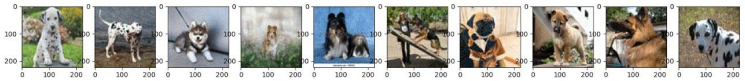
上圖為Autoencoder常見架構。下圖為我們使用MLP進行AutoEncoder與進行Decoder完後的結果。

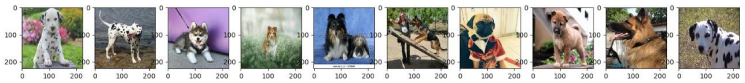




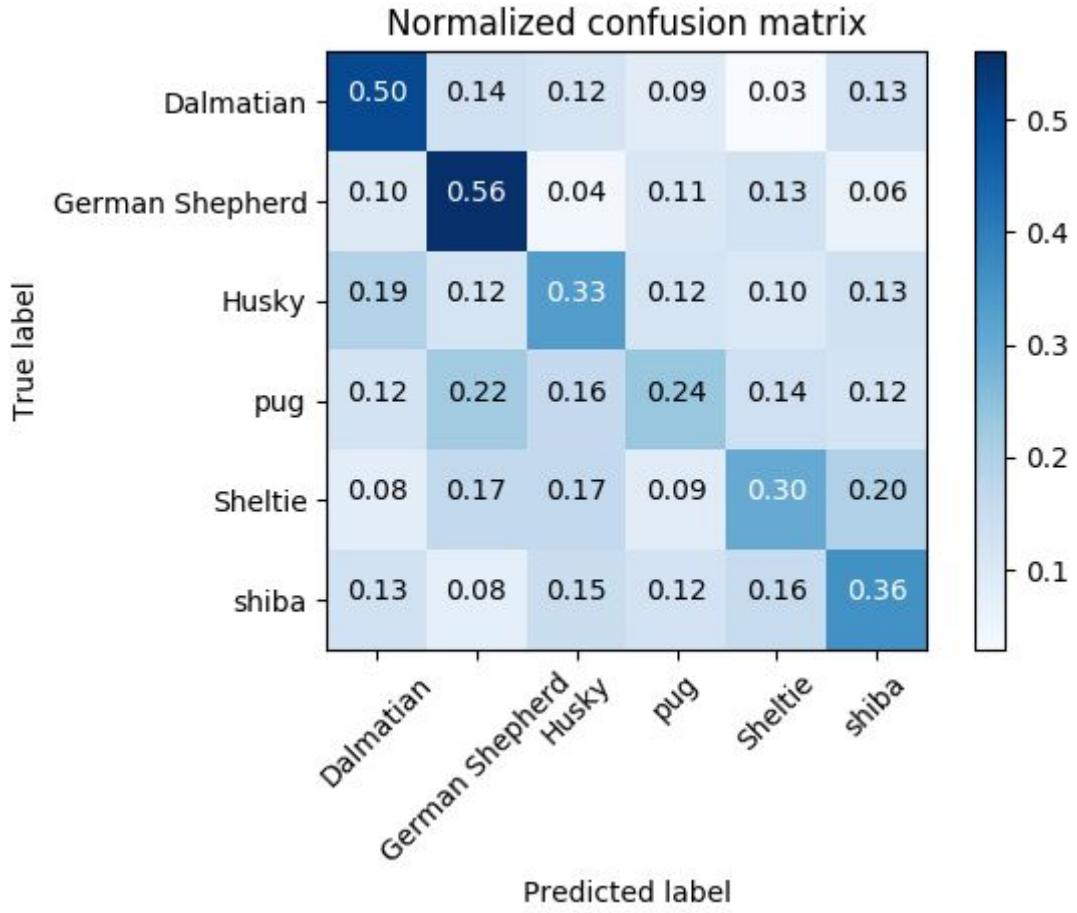
可以發現十分接近原始結果。下圖為使用Convolution Neural Networks進行encoder與decoder之結果。一樣也是十分接近原始圖片。



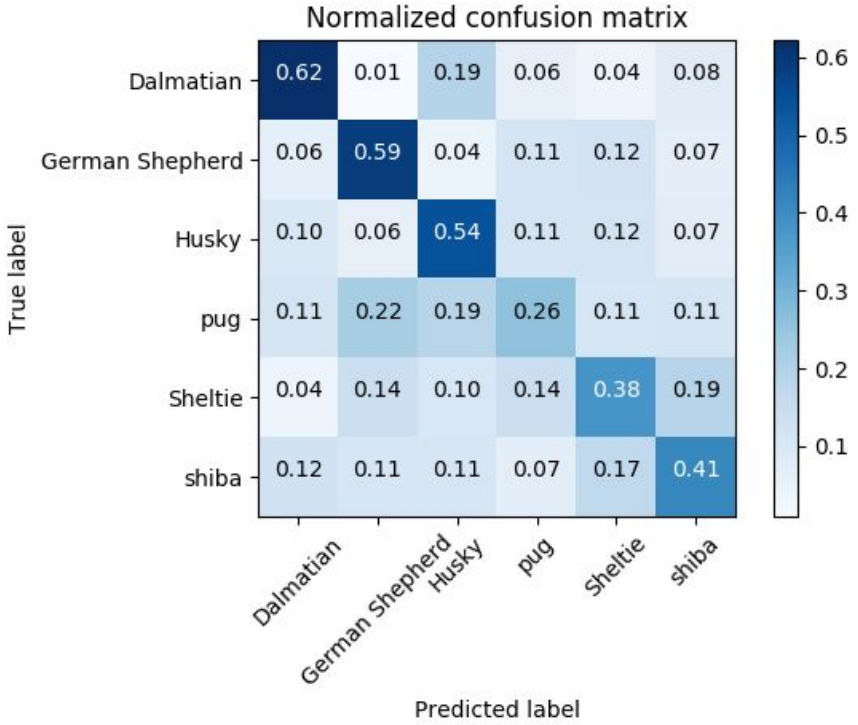




最後以SVM對test set預測時MLP版本準確度為38%，而CNN版本為46%。



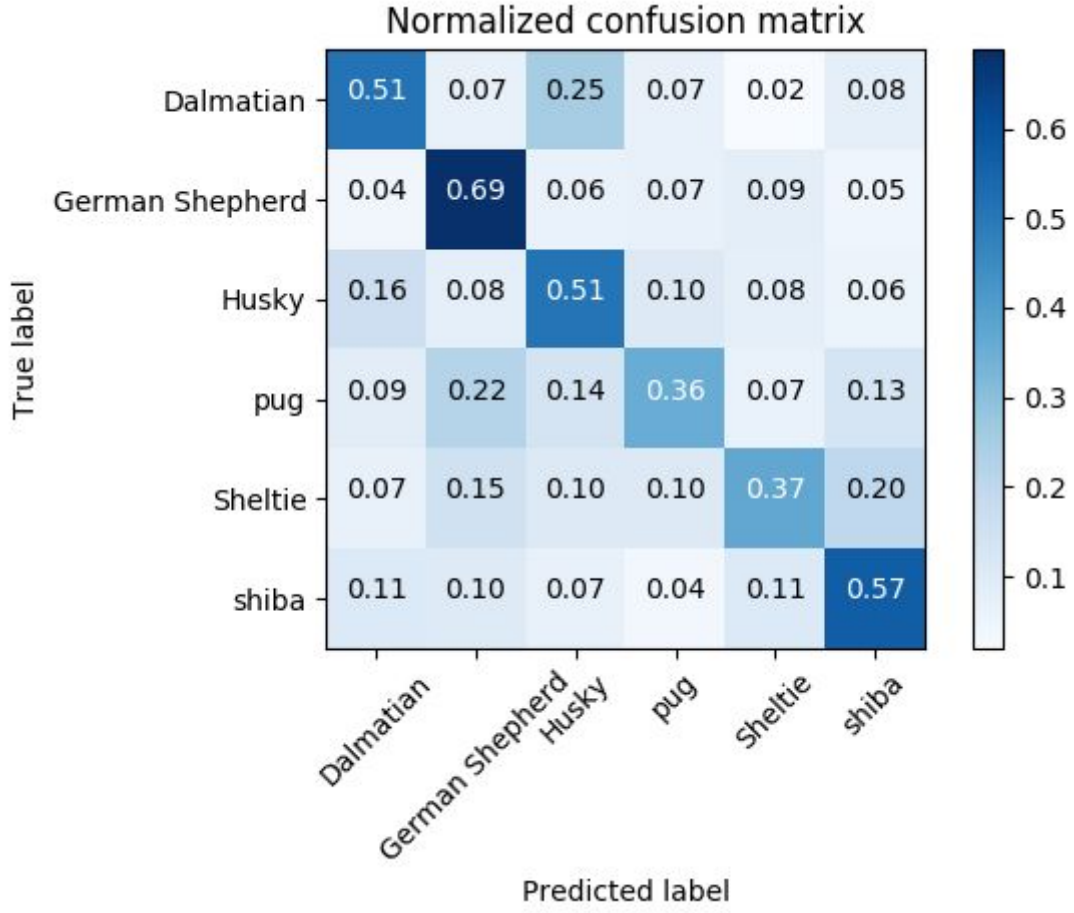
上圖為AutoEncoder(MLP)+SVM之Confusion matrix結果。



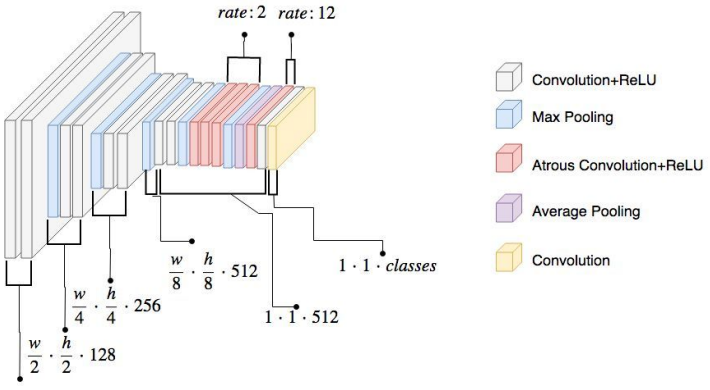
上圖為AutoEncoder(CNN)+SVM之confusion matrix。

E. Multilayer perceptron

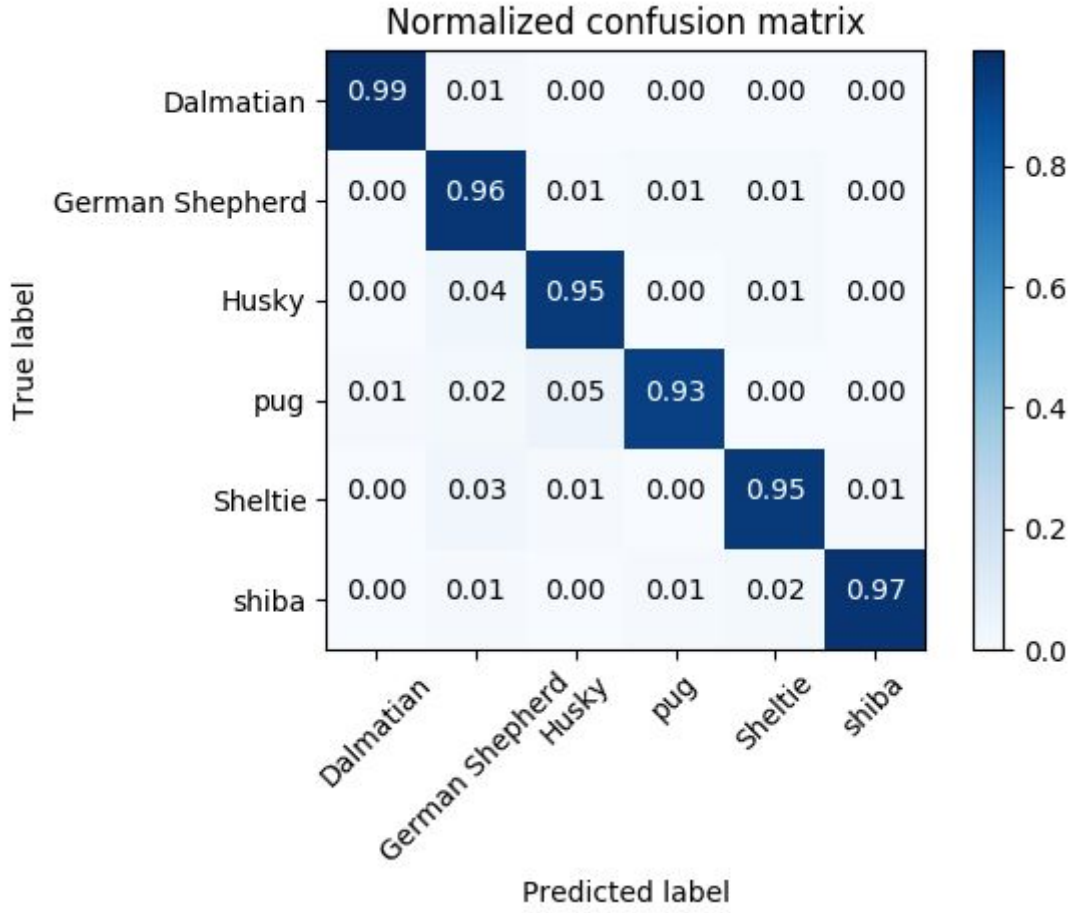
Multilayer perceptron部份是用tensorflow，是用一層256個neuron的hidden layer，同時Activation function使用SeLu，輸入使用64X64 的圖片進行訓練。Loss function使用cross entropy，其他優化使用Adam optimizer調整learning rate，同時還有進行alpha drop out、learning rate decay、learning rate schedule與weight regularization。最後在使用Learning rate為0.001，Batch大小為16下在Epoch 738跑出validation set 準確度53.98%，test set準確度50.66%，以下為此結果下的confusion matrix。



F. VGG16



VGG16是常見的CNN，我們是參考deeplab largeFov的VGG16架構和參數，並使用Tensorflow實做，但他們是將VGG16用在semantic segmentation上，我們是用在classification上，所以將最後一個average pooling做更改，使得w*h=1，然後最後的convolution層的class數調成6個，並使用pretrain weight在image net上，然後將卷積層的learning rate調成比分類層小0.5倍來進行finetune。值得一是，這個架構的VGG16使用的是atrous convolution，也就是帶洞卷積，利用padding零的方式使得kernel size變大，讓網路看的更大範圍的特徵，確不會增加太多運算成本。

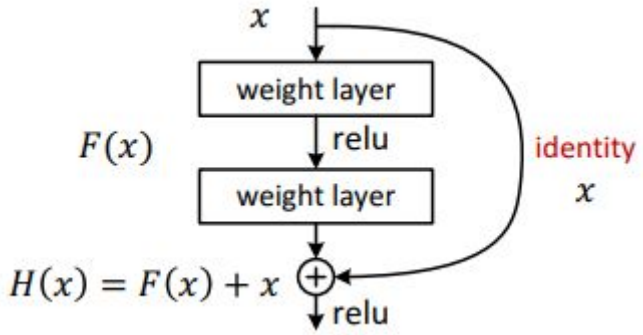


G. ResNet

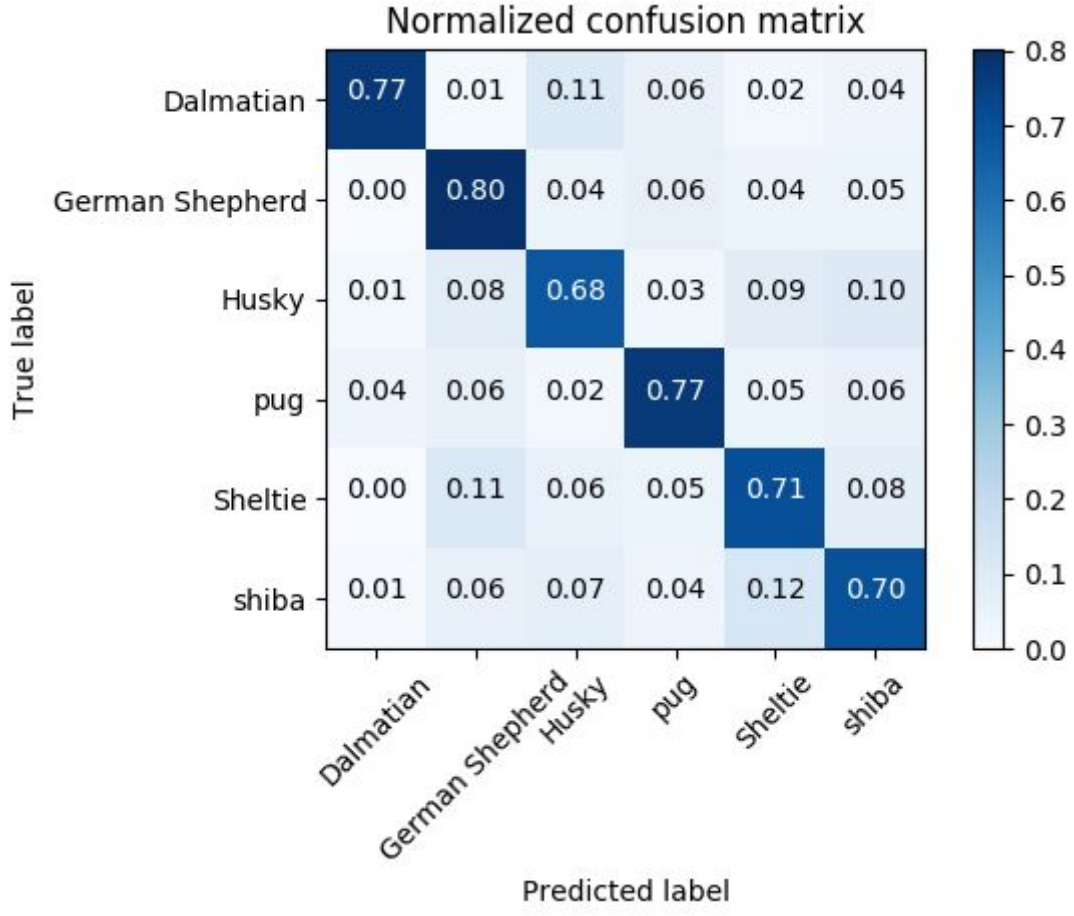
使用tensorflow進行撰寫之常見CNN架構，與VGG16的不同處是ResNet在block上會有short cut，因此比較不會遇到gradient vanish問題，因此網路可以疊的更深，並且以32X32大小的圖片進行訓練與測試，使用pretrain weight在cifar10上，對此我們採用了三種子架構：

1. Resnet-20
2. Resnet-56
3. Resnet-101

Loss function為cross entropy，convolution layer的learning rate為0.03，classification layer的learning rate為0.1，來進行finetune，Batch大小為128，最後使用Momentum optimizer和batch normalization進行優化，從實驗結果可以看出，若使用較小的圖片進行訓練且資料集不大可以加快收斂和訓練時間但是在down sample的過程中也會損失很多資訊，因此在最終的表現上只用76.32%，但我們將該網路訓練在cifar10上則可以得到91%的表現。

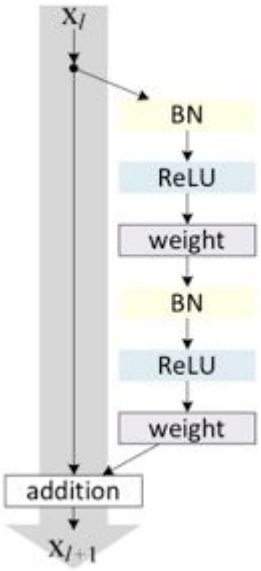


Neural Network	Best val accuracy	Epoch	Test accuracy
Resnet-20	79.25%	117	76.32%
Resnet-56	81.17%	155	74.86%
Resnet-101	82.44%	69	75.92%



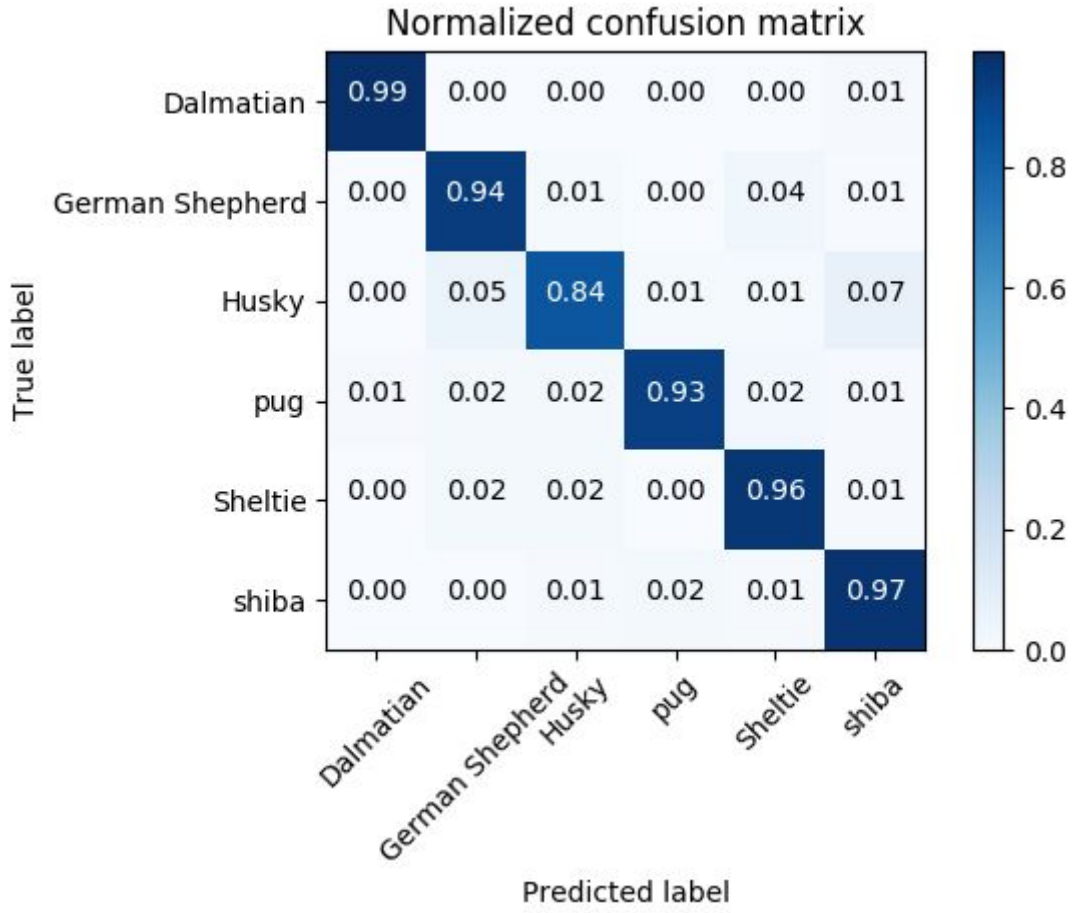
上圖為最佳表現model Resnet-20之結果confusion matrix。

H. ResNet-preact



Resnet-preact為另一常見CNN架構，與普通的ResNet的差異是將ReLu和batch normalization移到convolution前，並且一個block變成有3個卷積而不是2個，但是一個block的參數量仍維持不變，所以總體來講所需的參數量較普通ResNet少，此法我們使用256X等比例縮放之圖片進行訓練與測試，表現上明顯比使用32X32的ResNet好有93.65%，雖然仍比VGG16差，但是所需的參數量比起VGG16所需的還少的多。

Neural Network	Best val accuracy	Epoch	Test accuracy
Resnet-preact-50	95.21%	148	93.65%
Resnet-preact-101	94.15%	100	92.46%
Resnet-preact-152	94.14%	188	92.32%



上圖為最佳結果Resnet-preact 50之confusion matrix。

三、結果

在SVM上我們使用autoencoder和PCA進行降維只擷取重要特徵，雖然autoencoder有用到CNN但是，最後在表現上仍與PCA差不多，是因為我們將SVM和autoencoder分開訓練，所以autoencoder仍然不受label影響，若我們將autoencoder和SVM直接一起訓練相信CNN就可以發揮他的長處了，但這時也不需要decoder了只需要encoder就好了，也就是普通的CNN+SVM了。

在MLP方面可以發現MLP與CNN的決定性差異的一點就是，MLP比起CNN更容易受到input size的影響，假設我們的hidden layer為256用input size為224*224*3的大小和另一input size為64*64*3來比較，一個代表第一層的參數量有224*224*3*256和另一個代表64*64*3*256，可以看出參數量多的非常多，這也是我們選擇使用64*64*3的原因，但在表現上64*64*3與224*224*3差不多，且訓練時間更快。而在CNN上因為有sharing weights的特性所以比較不受影響。而另一項與CNN的差異是CNN可以擷取特徵的能力很強，而MLP在這方面相對弱勢，因此在較不簡單的資料集(ex: 非mnist)上MLP常常只用來做最後的分類層，這也是為什麼我們的MLP的表現只有50.66%。

在CNN方面從實驗結果得知，雖然使用較小的圖片可以加快訓練速度，但所得到的表現也較差，且愈深的網路並不代表愈好，若使用較簡單的資料集或是任務則使用較淺的網路即可，像是LeNet或是Alex Net也許都可以達到不錯的準確度，因為較深的網路往往代表要訓練的參數愈多，因此訓練時間可能較長且容易overfitting或是遇到gradient vanish的問題，但有些問題可以用early stopping或是short cut來減緩。

最後結果比較如下圖。

Models	Best val accuracy	Epoch	Test accuracy
PCA(RGB,50 component)+SVM	X	X	45%
AutoEncoder(MLP) +SVM	X	299	38%
AutoEncoder(CNN) +SVM	X	20	46%
MLP	53.98%	738	50.66%
VGG16	97.6%	233	95.76%
Resnet-101	82.44%	69	75.92%
Resnet-preact-50	95.21%	148	93.65%

四、貢獻

戴宏周:Data visualization, PCA, SVM
呂承祐:Data augmentation, MLP, Autoencoder, VGG16, ResNet
顏靖剛:Data collection, Data pre-proccessing

五、環境

戴宏周：

- OS: Linux (4.14.74-1-lts SMP x86_64 GNU/Linux)
- Distribution: Arch Linux
- python version: 3.7.0
- matplot: 2.2.3
- sklearn: 0.19
- pandas: 0.23
- tensorflow: python-tensorflow 1.11.0-2
- g++ 8.2.1

顏靖剛：

- OS: Linux (GNU/Linux 4.15.0-36-generic x86_64)
- Distribution: Ubuntu 16.04.5

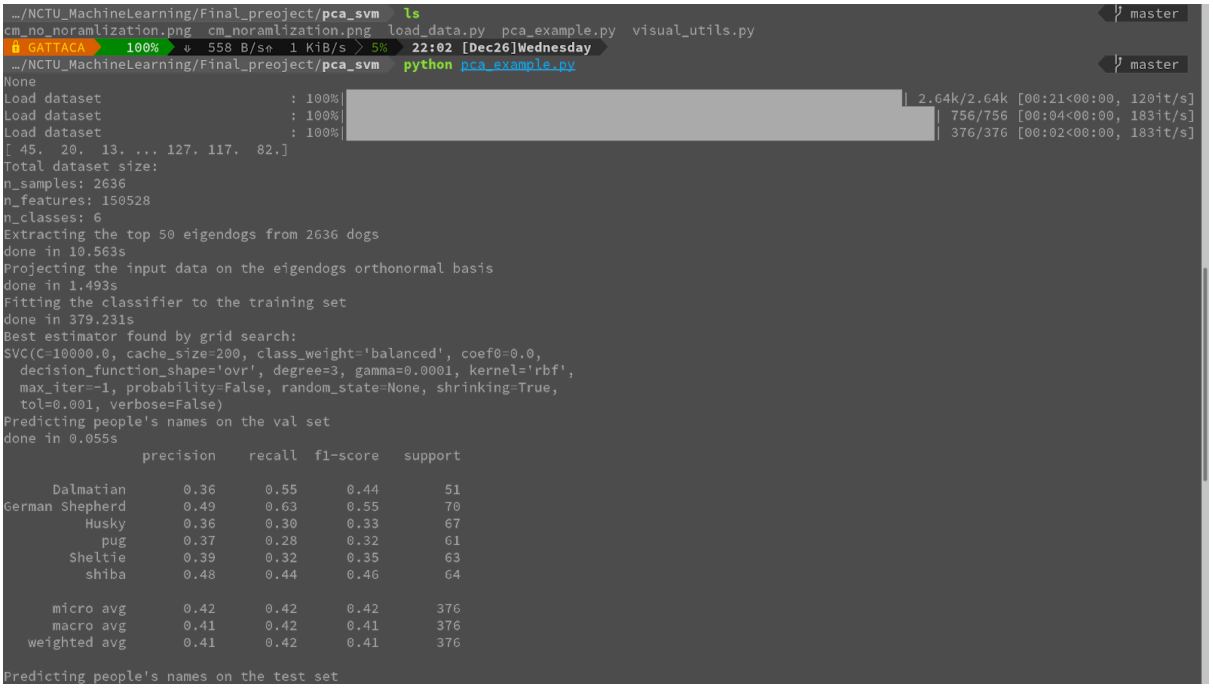
- python version: 3.5.2
- matplot: 3.0.0
- sklearn: 0.20.0
- pandas: 0.23.4

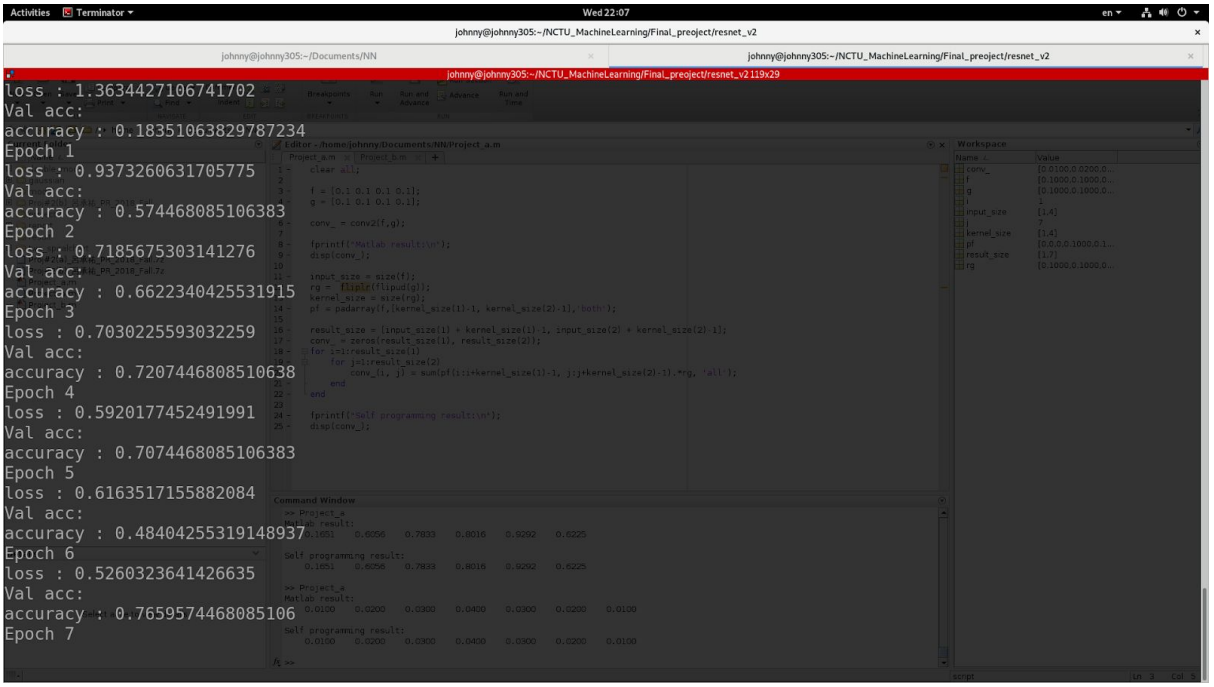
呂承祐：

- OS: Linux 4.18.10-arch1-1-ARCH
- Distribution: Arch Linux
- python version: 3.6.0
- matplot: 2.2.3
- sklearn: 0.20.0
- pandas: 0.23.4
- tensorflow-gpu 1.10.1
- g++ 8.2.1

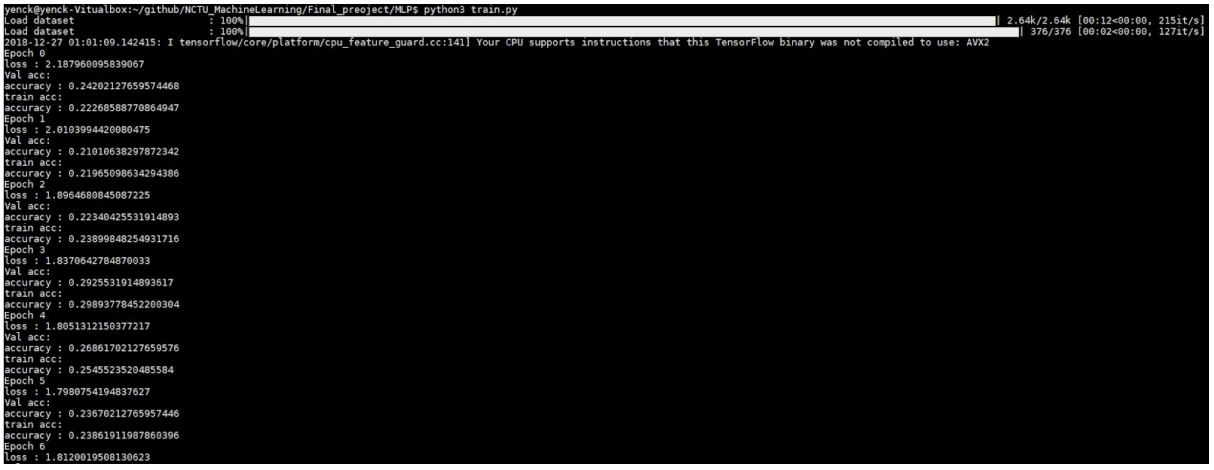
六、運行截圖

戴宏周




loss : 1.3634427106741702
Val acc:
accuracy : 0.18351063829787234
Epoch 1
loss : 0.9373260631705775
Val acc:
accuracy : 0.574468085106383
Epoch 2
loss : 0.7185675303141276
Val acc:
accuracy : 0.6622340425531915
Epoch 3
loss : 0.7030225593032259
Val acc:
accuracy : 0.7207446808510638
Epoch 4
loss : 0.5920177452491991
Val acc:
accuracy : 0.7074468085106383
Epoch 5
loss : 0.6163517155882084
Val acc:
accuracy : 0.48404255319148937
Epoch 6
loss : 0.5260323641426635
Val acc:
accuracy : 0.7659574468085106
Epoch 7

顏靖剛


yenck@yenck-Vitualbox:~/github/NCTU_MachineLearning/Final_preoject/MLP$ python3 train.py
Load dataset : 100%| 2.64k/2.64k [00:12<00:00, 215it/s]
Load dataset : 100%| 376/376 [00:02<00:00, 127it/s]
2018-12-27 01:01:09.142415: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
Epoch 0
loss : 2.187960095839067
Val acc:
accuracy : 0.24202127659574468
train acc:
accuracy : 0.22268588770864947
Epoch 1
loss : 2.0103994420080475
Val acc:
accuracy : 0.21010638297872342
train acc:
accuracy : 0.21965098634294386
Epoch 2
loss : 1.8964680845087225
Val acc:
accuracy : 0.22340425531914893
train acc:
accuracy : 0.23899848254931716
Epoch 3
loss : 1.8370642784870033
Val acc:
accuracy : 0.2925531914893617
train acc:
accuracy : 0.29893778452200304
Epoch 4
loss : 1.8051312150377217
Val acc:
accuracy : 0.26861702127659576
train acc:
accuracy : 0.2545523520485584
Epoch 5
loss : 1.7980754194837627
Val acc:
accuracy : 0.23670212765957446
train acc:
accuracy : 0.23861911987860396
Epoch 6
loss : 1.8120019508130623