FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Universidade de Coimbra

Aprendizagem Profunda Aplicada

2021/2022

Practical Assignment 2

José Pedro Araújo Azevedo
uc2016236736@student.uc.pt

Rafael Alves Vieira
uc2017257239@student.uc.pt

9 de dezembro de 2021

1. Introdução

Este relatório descreve as análises e conclusões obtidas na realização trabalho prático nº2, que consiste na implementação de uma *CNN* envolvendo uma network *VGG16* para a classificação de imagens do dataset CIFAR10. Ainda uma implementação do algoritmo YOLOv3 com imagens do KITTI subset preparado pelos docentes para este trabalho.

2. Object Classification with Transfer Learning Techniques

Como o treino desta rede é bastante demorado, decidiu-se executar o treino no *hardware* pessoal, pelo que comparando com o *Google Collab* os resultados obtidos em *hardware* pessoal foram mais vantajosos em termos de tempo e com pequenas melhorias nos valores de *accuracy*.

2.1. Cifar10 com network *VGG16* from scratch

Utilizando os melhores hiperparâmetros obtidos no primeiro Assignment, correu-se então esta network de raíz com os seguintes hiperparâmetros:

- *Transform* com *resize* de 64 e normalização com *mean*=[0.4915, 0.4823, 0.4468] e *std*=[0.2470, 0.2435, 0.2616].
- Batch size = 64.
- Learning Rate inicial de 0.01, aplicando o *scheduler* = *ReduceLROnPlateau*.
- Número de *epochs* = 50.
- Otimizador = *Stochastic gradient descent*.

Resultados obtidos

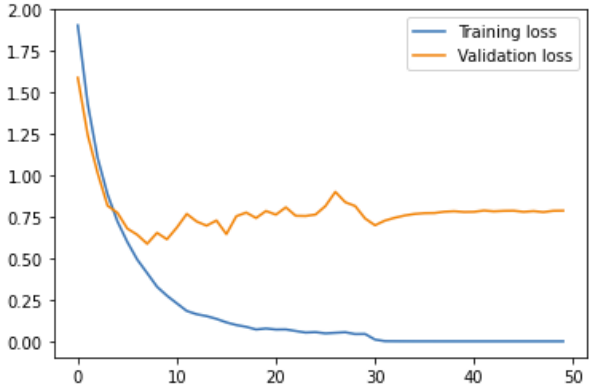

Figura 1: Loss and Validation curves.

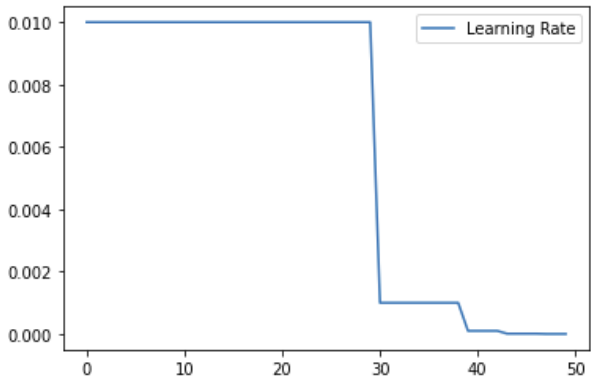

Figura 2: Learning Rate variation.

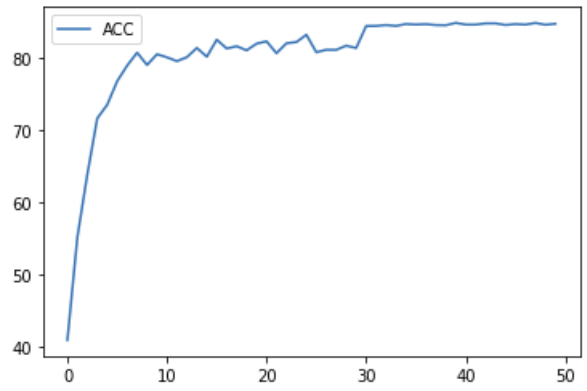

Figura 3: Train accuracy curve

```
Accuracy using 10000 test images: 84.75 %

Accuracy of airplane : 86 % in 995 Images
Accuracy of automobile : 93 % in 944 Images
Accuracy of bird : 74 % in 982 Images
Accuracy of cat : 73 % in 992 Images
Accuracy of deer : 83 % in 1005 Images
Accuracy of dog : 76 % in 1014 Images
Accuracy of frog : 87 % in 1035 Images
Accuracy of horse : 87 % in 1024 Images
Accuracy of ship : 92 % in 1013 Images
Accuracy of truck : 92 % in 996 Images
```

Figura 4: Accuracy for each class.

Com este treino obteve-se um *mCA* total de 84.76%. Tempo de execução : 3h5minutos.
Como foi afirmado pelo docente da cadeira, só se vai fazer combinação de hiperparâmetros para a versão com melhores resultados.

2.2. Cifar10 com network *VGG16* pré-treinada

Nesta secção a única diferença implementada foi usar o modelo *VGG16* com *pretrained = True*, e ainda reduzir o número de *epochs* de 50 para 30, como o modelo já foi pré-treinado manter o mesmo número de *epochs* não será muito vantajoso em termos de resultados e tempo.
De salientar que as métricas utilizadas foram iguais ao ponto *2.1.*

Resultados obtidos

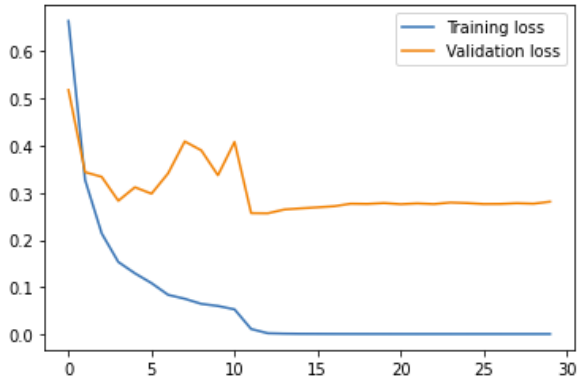

Figura 5: Loss and Validation curves.

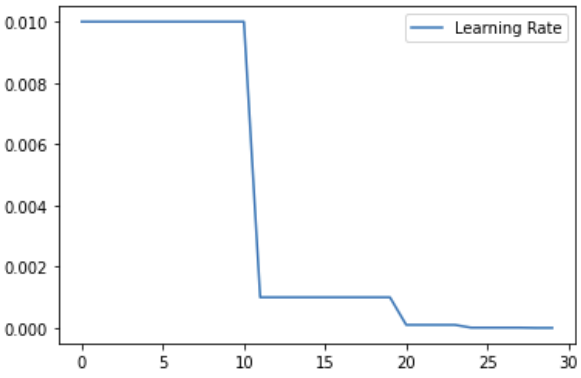

Figura 6: Learning Rate variation.

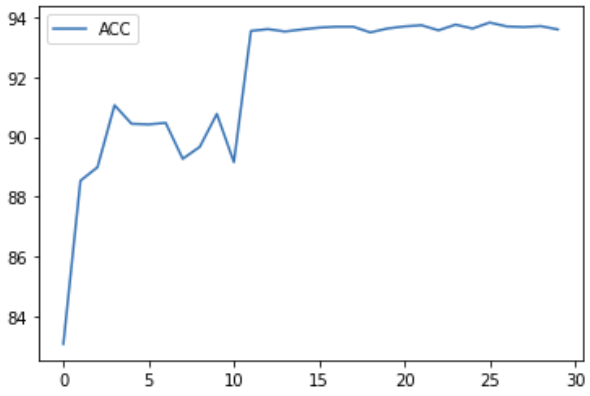

Figura 7: Train accuracy curve

```
Accuracy using 10000 test images: 93.73 %

Accuracy of airplane : 95 % in 968 Images
Accuracy of automobile : 95 % in 1031 Images
Accuracy of bird : 92 % in 968 Images
Accuracy of cat : 87 % in 1006 Images
Accuracy of deer : 93 % in 955 Images
Accuracy of dog : 88 % in 1004 Images
Accuracy of frog : 97 % in 967 Images
Accuracy of horse : 95 % in 1050 Images
Accuracy of ship : 96 % in 1044 Images
Accuracy of truck : 96 % in 1007 Images
```

Figura 8: Accuracy for each class.

Com este treino obteve-se um *mCA* total de 93.71%. Tempo de execução : 54 minutos.

2.3. Cifar10 com network *VGG16* pré-treinada, partindo de um ponto

Nesta parte do trabalho pretende-se treinar a *network* partindo de um *starting point*. Na tentativa de obter resultados melhores ou "intermédios"comparando com as duas versões da *network* mencionadas acima, pode-se tentar modificar as camadas do *classifier*, fazer alterações no dataset, modificar hiperparâmetros, e obter resultados melhores em termos de accuracy e tentar reduzir o tempo de treino. As modificações que foram aplicadas, apenas se vai abordar uma pequena parte, porque no entanto houve grande parte de alterações que não trouxeram melhores resultados comparando com a versão *from scratch*. De salientar que as modificações mencionadas em baixo, podem ser consultadas em melhor pormenor nos *notebooks* em anexo.

Modificações

1 - Pré-processamento dos dados, alteração do *Classifier*.

Aplicou-se no pré processamento:

- Resize de 224, visto que a *network* tem este input.
- Random Rotation(5).
- Random Horizontal Flip(0.5).
- Random Crop(224,padding=10).

O *Classifier* sofreu alterações, a primeira *layer* de *output_features* foi aumentada de *4096* para *8192* e adicionou-se outra *layer* com *8192* e *4096* de *input_features* e *output_features* respetivamente. As duas últimas camadas foram mantidas, com a adição de *LogSoftmax(dim = 1)*.
Nas restantes camadas foram aplicadas *freeze*. Nesta tentativa o número de *epochs* foi reduzido para 25.

Resultados obtidos:
mCA = 87.32%, o tempo de treino foi 2h15 minutos.

2 - Redução de *features* nas layers
Esta modificação foi idêntica à anterior, no entanto, em vez de aumentarmos o número de *features*, reduziram-se para *2048*. **Resultados obtidos:**
mCA = 88.44%, o tempo de treino foi 2 horas.

3 - Apenas pré-processamento alterado, sem *freeze*.
Pretende-se neste ponto, verificar as alterações de apenas generalizar o *dataset*, sendo assim partindo das alterações da *primeira modificação* aplicou-se apenas o pré-processamento indicado neste ponto.

Resultados obtidos: *mCA* = 94.43%, o tempo de treino foi 5h20 minutos. Este ponto foi fulcral para retirar conclusões, visto que aumentar as imagens para *224 pixels* pode aumentar um pouco a *performance* do modelo, mas em contrapartida aumentar o tempo do treino.
Comparado com a *network* pré treinada com apenas o *resize = 64* esta alteração de 0.7% em *accuracy* não se tira muita vantagem em termos de tempo de treino.

4 - Redução de *layers* e resize = 64
Utilizaram-se as mesmas *layers* indicadas no pronto 2, no entanto com *resize = 64* nas imagens.
Neste ponto tambem se aumentou a *learning rate* para 0.05. Com o intuito do treino ser bastante rápido

Resultados obtidos: *mCA* = 71.34%, o tempo de treino foi de 22 minutos .

5 - Modificação do pré-processamento.
Neste ponto manteve-se as *layers* indicadas no ponto *2)*, fazendo mais *Data Augmentation*, nomeadamente fazer *zooms* nas imagens e mudar *shear angles* e ainda modificar brilho,contraste e saturação das imagens. *mCA* = 73.92%, o tempo de treino foi 30 minutos.
Como estas alterações influenciam os valores de *accuracy* e conseguem generalizar mais o modelo, evitanto possivelmente *overfitting* achou-se importante documentar esta parte, sendo que poderia-se implementar estas alterações de *Data Augmentation* no modelo sem *freeze* para tentar obter melhores resultados.

Conclusões
Infelizmente não se obteve resultados melhores manipulando o *Classifier*, tanto em termos de *accuracy* do modelo final, como em termos de tempo de execução, o que achamos uma métrica bastante importante.
De salientar que fizeram-se outras alterações na *network* para além das mencionadas, no entanto como os resultados não foram benéficos, não foram documentados.

Melhores resultados obtidos, alínea *2*

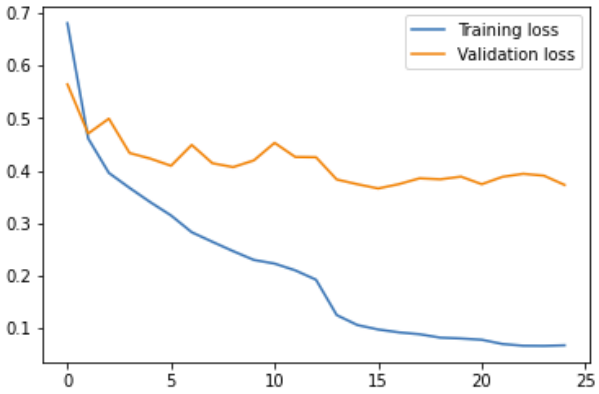

Figura 9: Loss and Validation curves.

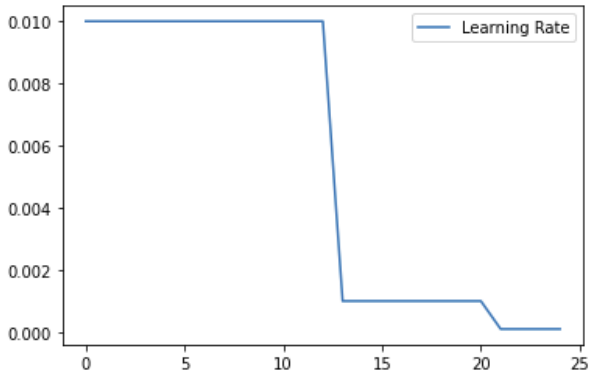

Figura 10: Learning Rate variation.

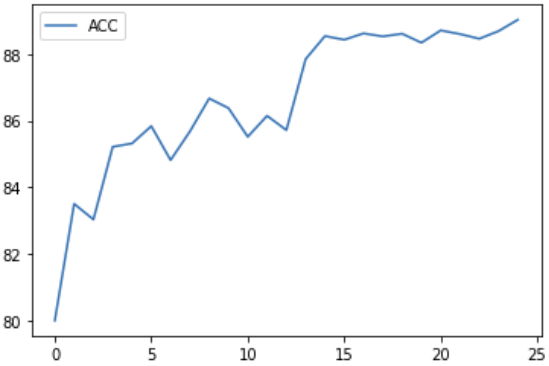

Figura 11: Train accuracy curve

```
Accuracy using 10000 test images: 88.37 %

Accuracy of airplane : 91 % in 971 Images
Accuracy of automobile : 94 % in 990 Images
Accuracy of bird : 85 % in 1013 Images
Accuracy of cat : 77 % in 1071 Images
Accuracy of deer : 86 % in 1002 Images
Accuracy of dog : 82 % in 959 Images
Accuracy of frog : 90 % in 994 Images
Accuracy of horse : 90 % in 994 Images
Accuracy of ship : 92 % in 980 Images
Accuracy of truck : 92 % in 1026 Images
```

Figura 12: Accuracy for each class.

2.4. Extra: *VGG16 versus ResNet18*

Também implementou-se o modelo *ResNet18*, nesta *network* não se modificou nada, ou seja, estava apenas pré treinado.

Arquitetura da rede

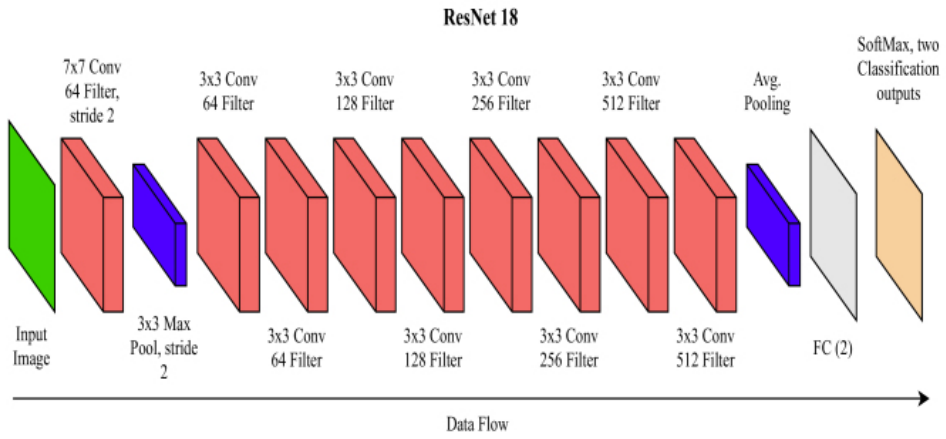

Figura 13: Ilustração da arquitetura *ResNet18*

Foi executado inicialmente com os parâmetros:

- Batch size = 64.
- Número de *epochs* = 25.
- *Learning Rate inicial = 0.01*.
- Otimizador SGD.

De salientar que foi aplicado o pré-processamento mais " *completo* "efetuado até esta alínea, nomeadamente:

- Resize de 224.
- Random Rotation(5).
- Random Horizontal Flip(0.5).
- Random Crop(224,padding=10).
- Random Affine(0, shear = 10, scale = (0.8,1.2)).
- Color Jitter (0.2)

Neste treino obteve-se um *mCA* de 95.47% o qual demorou 2 horas e 1 minuto a executar.

2.4.1. Combinação de hiperparâmetros na melhor rede.

Apesar de ter-se gasto mais tempo sob a *network VVG16*, obtiveram-se melhores resultados na *Resnet18*. Para tal, irá-se combinar alguns hiperparâmetros na tentativa de obter melhores resultados.

	Accuracy Testing 1000 Images | F1 - Score									
Optimizer	**SGD**						**ASGD**		**SGD**	
Epochs	**25**				**15**		**25**		**25**	
Learning Rate	**0.01**				**0.001**		**0.01**		**0.01**	
Batch Sizes / Classes	**32**		**64**		**64**		**64**		**128**	
Airplane	96	0.96	94	0.95	94	0.95	95	0.96	96	0.96
Automobile	96	0.97	96	0.98	97	0.96	97	0.98	96	0.98
Bird	92	0.94	94	0.95	94	0.93	94	95	95	0.95
Cat	91	0.90	90	0.91	89	0.90	90	0.91	92	0.90
Deer	96	0.97	96	0.96	94	0.94	96	0.96	95	0.96
Dog	91	0.91	92	0.92	89	0.91	92	0.92	90	0.92
Frog	96	0.97	96	0.97	96	0.96	97	0.97	97	0.97
Horse	95	0.96	96	0.97	96	0.95	96	0.96	95	0.97
Ship	97	0.97	98	0.97	96	0.97	97	0.98	97	0.98
Truck	96	0.97	98	0.97	95	0.97	96	0.97	97	0.97
mCA	**95.15 %**		**95.47 %**		**94.57 %**		**95.45 %**		**95.44 %**	
Runtime	**2 horas 12 minutos**		**2 horas 1 minuto**		**1 hora 10 minutos**		**1 hora 54 minutos**		**2 horas 2 minutos**	

Figura 14: Combinação de hiperparâmetros, *ResNet18*

Testou-se para 25 *epochs* e *learning rate* inicial de 0.01 com *batches* de 32 e 64, como o modelo melhor foi o de 64 testou-se para *learning rate* reduzida para 0.001 diminuindo o número de *epochs* para 15.Como o modelo melhor desta comparação foi usando *epochs* de 25 e *learning rate* $= 0.01$, fez-se o mesmo treino usando um diferente *optimizer* , o *Averaged Stochastic Gradient Descent*. Eventualmente correu-se o treino para *batches* de 128.

Gráficos dos melhores resultados obtidos

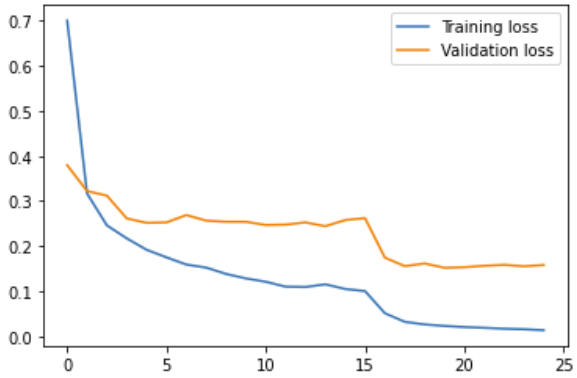

Figura 15: Loss and Validation curves.

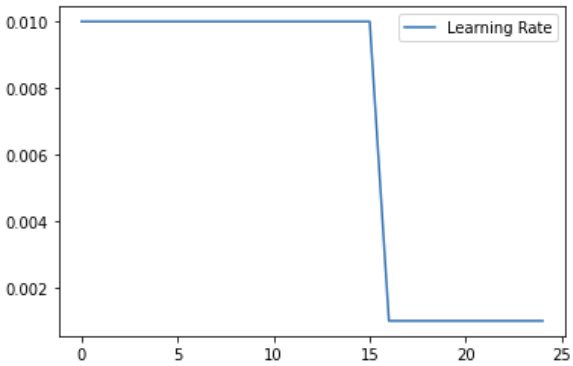

Figura 16: Learning Rate variation.

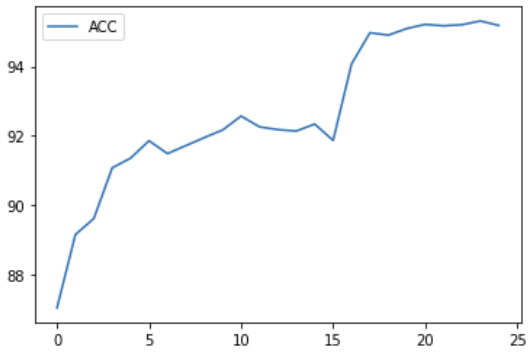

Figura 17: Train accuracy curve

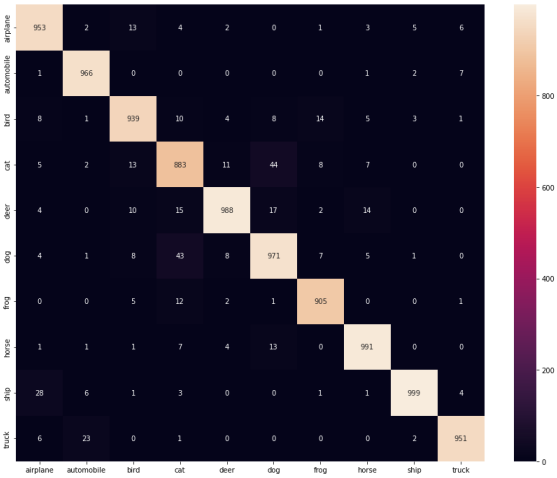

Figura 18: Confusion matrix.

2.5. Comparação entre treino de raíz e *transfer learning*.

Usando *transfer learning* o treino fica com melhores resultados, mais generalizado e ainda mais rápido. Portanto podemos afirmar que em certos casos, nomeadamente quanto há um dataset maior, usar *transfer learning* trás mais vantagens.

2.6. Respostas , no que diz respeito ao *transfer learning*.

Why transfer learning is important in deep learning models?

Transfer Learning trás bastantes beneficios para o desenvolvimento de modelos, nomedamente o tempo de execução, e a *accuracy* do modelo.
Enquanto por exemplo treinar um modelo de raíz demoraria 3 horas com um *mCA* de 86%, com *transfer learning* consegue-se um modelo mais generalizado com cerca de 94% de *mCA* demorando apenas 50 minutos(Esta comparação foi baseada nos pontos *2.1* e *2.2*, considerando *GPU's* iguais).

What are the main differences that you found relevant about the use of transfer learning techniques?

Pode-se ter um *dataset* reduzido e mesmo assim obter bons resultados.
Normalmente em *transfer learning* as ultimas *layers* é que são substituidas e estas são inicializadas com *random weights*, as outras *layers* podem ser levar *freeze* ou não, ou seja podem-se treinar ou não.
Em *learning from scratch* os pesos são inicializados e treina-se então o dataset, demorando mais tempo.

Can you further improve the performance of your best CNN architecture? Explain your answer

Sim é possível melhorar a performance do modelo, no entanto temos que considerar um limiar entre *accuracy* do modelo e tempo/recursos gastos.
Consideramos os seguintes tópicos importantes para melhorar a performance da arquitetura:

- Aumentar o dataset.
- Normalizar os dados.
- Usar Data Augmentation, por exemplo(ajustar tamanhos das imagens, implementar zooms, mudar cores, cortar imagens) assim o modelo fica mais generalizado evitando *ovetfitting*.
- Modificar hiperparâmetros(*Learning Rate*, *epochs*,*batch sizes* por exemplo).
- Aumentar/diminuir *layers*.
- Técnicas de regularização(modificar *dropouts*).

3. Object Detection

You Only Look Once

De modo a contextualizar a explicação dos resultados obtidos damos uma breve explicação do nosso entendimento do funcionamento do algoritmo.
YOLO é um algoritmo usado em redes neuronais convolucionais para a deteção de objetos em imagens. Neste trabalho foi usado para a deteção de carros. Ao contrário de muitos outros algoritmos, YOLO tem o seu nome de *you only look once* visto que consegue detetar os objetos correndo o algoritmo

apenas uma vez (olha apenas uma vez para a imagem). A CNN é usada para prever as probabilidades das classes e as *bounding boxes* em simultâneo. O output da rede é um vetor que tem a informação acerca da classe a que pertence (número 0 se for um carro por exemplo), e os valores das coordenadas da bounding box associada a essa classe, x e y, bem como a sua largura e altura w e h. Isto é escalável para o número de objetos na imagem.

Na primeira layer como a grid size é mais pequena podemos detetar objetos grandes, na segunda layer objetos de tamanhos médios e na ultima camada objetos de pequenas dimensões.
Esta analogia de grids pode ser ilustrada na seguinte imagem:

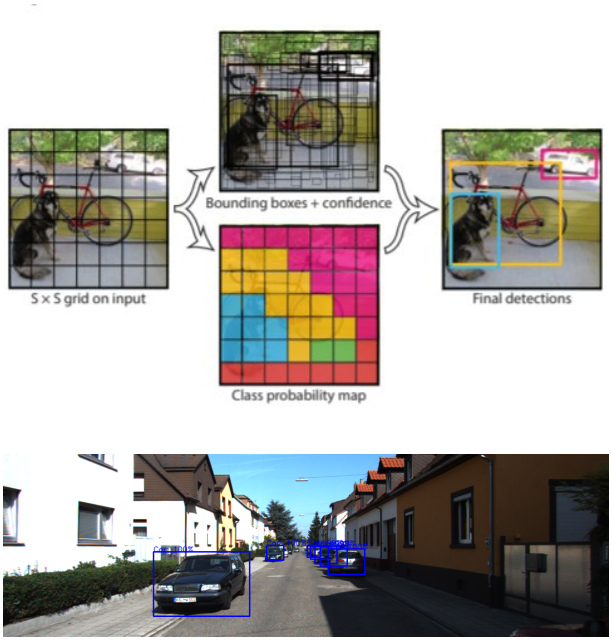

Figura 19: Imagem de um resultado obtido. Em 3 layers diferentes teríamos esta imagem dividida em células com grid sizes diferentes em cada layer.

Em diferentes células da grid, associamos a probabilidade de pertencer a uma classe. Na imagem que tem as várias bounding boxes aplicamos supressão não máxima e associamos as bounding boxes resultantes a uma classe atravez da probabilidade da zona a que pertença.
Quanto maior for o grid size mais dividida fica a imagem e mais features temos, e conseguimos detetar objetos mais pequenos, porque poderíamos no caso de uma grid size mais pequena, ter uma feature que apanhasse um objeto pequeno inteiro, e assim provavelmente nem o iríamos conseguir identificar. Mas com grid size maior, essa feature é dividida em pequenas features, e conseguimos identificar o objeto porque passamos a ter mais features com alta probabilidade de pertencerem ao objeto, e não apenas uma.

Transfer learning vs Training from Scratch

Usando os resultados com os pesos que obtivemos a treinar de raiz, obtivemos um mAP = 0.54 (e 0.54 de AP para a classe do carro, visto que só fizemos para 1 classe). E ao usar os pesos dados

pelos docentes obtivemos mAP = 0.71. Nos resultados em transfer learning como tivemos alguns problemas com os limites de utilização das gpus da google, ficámos com um checkpoint nas 175 épocas e resumimos depois o treino a partir daí, usando depois 25 épocas até acabar a fase de treino. Esse checkpoint de 25 épocas (que marca as 200) é que depois foi usado no algoritmo de teste. Em relação aos resultados nas imagens podemos observar resultados para duas imagens iguais, e verificamos que os pesos obtidos por transfer learning são melhores não só de acordo com as métricas mas também observando os resultados:

Figura 20: Treino de raíz

Figura 21: Transfer Learning

Os pesos obtidos por transfer learning vieram de outras apliações (COCO dataset) e como podemos observar foram bastante eficientes nesta aplicação. Transfer learning permite armazenar o estado de aprendizagem de uma rede, e isto é extremamente útil e conveniente. Nas imagens acima vemos que há bounding boxes que não estão nos carros (falsos positivos), e o carro mais pequeno do fundo não foi reconhecido, e como o carro mais pequeno é detetado por layers que tenham grid size maior, podemos prever até que os valores da loss serão mais altos nesta layer com os pesos treinados de raíz, visto que vai errar mais.

Métricas

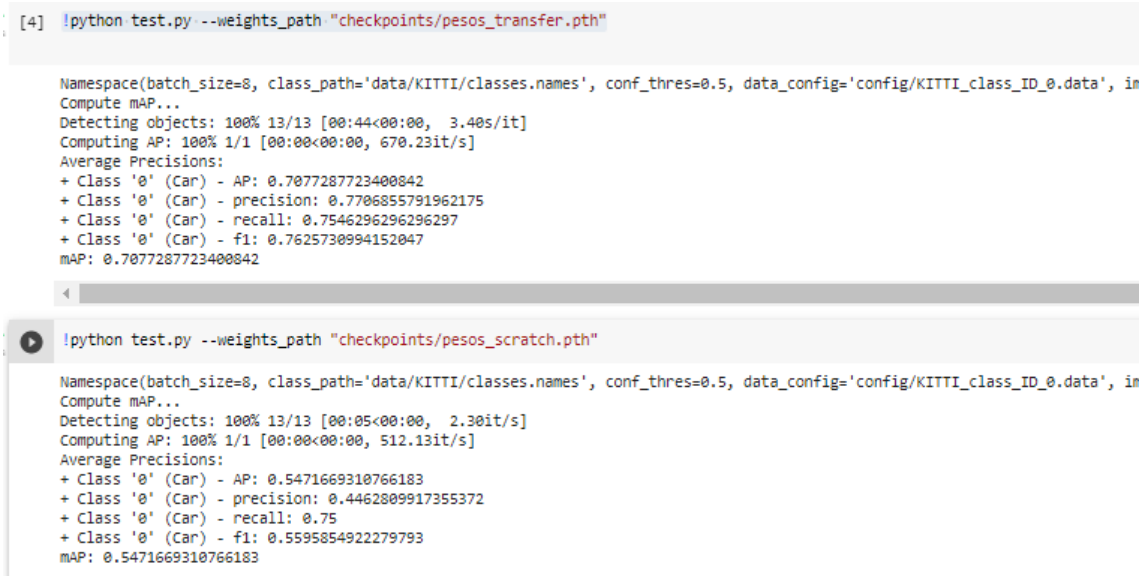

Figura 22: Métricas (transfer learning em cima e treino de raíz em baixo).

Treino de raíz

- AP 0.54
- Precision 0.45
- Recall 0.75
- F1 score 0.55

Treino com transfer learning (COCO)

- AP 0.71
- Precision 0.77
- Recall 0.75
- F1 score 0.76

Explicação da loss function

```
---- [Epoch 25/25, Batch 67/67] ----
+------------+--------------+--------------+--------------+
| Metrics    | YOLO Layer 0 | YOLO Layer 1 | YOLO Layer 2 |
+------------+--------------+--------------+--------------+
| grid_size  | 13           | 26           | 52           |
| loss       | 0.177279     | 0.223155     | 0.222541     |
| x          | 0.022184     | 0.031783     | 0.029178     |
| y          | 0.007749     | 0.006013     | 0.012956     |
| w          | 0.049091     | 0.020115     | 0.009977     |
| h          | 0.074244     | 0.027787     | 0.022855     |
| conf       | 0.024002     | 0.137458     | 0.147574     |
| cls        | 0.000009     | 0.000000     | 0.000001     |
| cls_acc    | 100.00%      | 100.00%      | 100.00%      |
| recall50   | 1.000000     | 0.952381     | 0.956522     |
| recall75   | 0.666667     | 0.809524     | 0.913043     |
| precision  | 1.000000     | 0.909091     | 0.758621     |
| conf_obj   | 0.987141     | 0.938188     | 0.943712     |
| conf_noobj | 0.000099     | 0.000254     | 0.000236     |
+------------+--------------+--------------+--------------+
Total loss 0.62297523021698
---- ETA 0:00:00
```

Com esta tabela de métricas que temos como output a cada batch durante o treino, podemos observar os resultados da loss function. Onde x y w e h são os ajustes feitos nas coordenadas e ao tamanho da bounding box. A total loss é calculada somando todas as losses.

A loss function é composta pelo ajuste das cordenadas o ponto central da anchor box, o ajuste dos valores da altura e largura, o ajuste do parametro de confiança e o ajuste da cls. O mesmo pode ser observado na imagem seguinte, retirada do código:

```
# Loss : Mask outputs to ignore non-existing objects (except with conf. loss)
loss_x = self.mse_loss(x[obj_mask], tx[obj_mask])
loss_y = self.mse_loss(y[obj_mask], ty[obj_mask])
loss_w = self.mse_loss(w[obj_mask], tw[obj_mask])
loss_h = self.mse_loss(h[obj_mask], th[obj_mask])
loss_conf_obj = self.bce_loss(pred_conf[obj_mask], tconf[obj_mask])
loss_conf_noobj = self.bce_loss(pred_conf[noobj_mask], tconf[noobj_mask])
loss_conf = self.obj_scale * loss_conf_obj + self.noobj_scale * loss_conf_noobj
loss_cls = self.bce_loss(pred_cls[obj_mask], tcls[obj_mask])
total_loss = loss_x + loss_y + loss_w + loss_h + loss_conf + loss_cls
```

Como melhorar os resultados obtidos

Se tivéssemos nós mesmos de melhorar os resultados obtidos não começaríamos por alterar a arquitetura da rede, nem hiper parãmetros, porque também não a entendemos a 100%. O pré-processamento das imagens também já é realizado e foi um pouco isso que fizemos no documento *datasets.py*. Verificámos que há um pré-processamento específico para a arquitetura (um resize para podermos partir a imagem mais facilemtne em grids, etc) portanto nesta fase do trabalho (algoritmo YOLO) não começariamos por mudar nada na aquitetura nem no pré-processamento dos dados.
Pondo isso de parte, vimos que obtemos melhores resultados ao usar transfer learning, portanto achamos que esse método seria a melhor aposta.
Aumentar a variedade do dataset e fazer data augmentation poderia trazer melhores resultados mas temos de ter em conta que o treino já demora imenso tempo com apenas 400 imagens, mas se conseguíssemos diminuir o tempo de treino com pesos vindos de transfer learning poderíamos considerar data augmentation.
No entanto não podemos negar que poderíamos obter resultados melhores ao alterar a arquitetura da rede e os hiper parâmetros se tivéssemos mais conhecimento na área, porque na verdade o que vai mudando entre as versões do algoritmo YOLO é a arquitetura da rede.
No âmbito de saber mais acerca deste assunto fizemos uma pesquisa nos métodos existentes para melhorar os resultados obtidos, e baseando-nos no seguinte site `https://blog.paperspace.com/improving-yolo/`. O método que cativou mais a nossa curiosidade foi *image mixup* onde é feita interpolação entre os valores dos pixeis de duas imagens para obter uma nova imagem com a informação das duas. Isto cria mais variadade nossas samples pode trazer melhores resultados, visto que é fácil, por exemplo, numa imagem que tenha uma cozinha, ter frigoríficos e micro-ondas, mesas e cadeiras na mesma imagem. Mas com image mixup podemos ter um elefante na nossa imagem da cozinha, melhorando o nosso algoritmo de deteção de objetos fora do ambiente típico onde poderíamos encontrar um elefente. Isto varia conforme casos de aplicação, o exemplo dado anteriormente sería útil se, por exemplo, usássemos um drone para procurar espécies ameaçadas de elefantes. No nosso caso de aplicação pordeiram ser usados carros no lugar dos elefantes.

Webgrafia

https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-
https://pytorch.org/
https://www.section.io/engineering-education/introduction-to-yolo-algorithm-for-object-
https://www.fatalerrors.org/a/detailed-explanation-of-yolov3-loss-function.html
https://blog.paperspace.com/improving-yolo/

Bibliografia

Deep Learning with PyTorch by Eli Stevens
K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition"
YOLOv3: An Incremental Improvement by Joseph Redmon, Ali Farhadi