ECI2019 - COMPETENCIA DESPEGAR
CLASIFICADOR DE IMÁGENES DE HOTELES

Leticia L. Rodríguez

July 20, 2019

RESUMEN

En el presente informe se detalla el proceso llevado a cabo para la elaboración de un clasificador de imágenes de hoteles en el contexto de las Competencias de Datos de la ECI 2019.
Repositorio en:`https://github.com/letyrodridc/competenciaECI2019` [privado]

***K*eywords** Clasificación de Imágenes · Tranfer Learning · Redes Convolucionales · Optimización de Redes Neuronales

1 Preparación de los datos

El pre-procesamiento de datos se vió relacionado con el framework elegido, Pytorch sobre Tensorflow.

A diferencia de Keras, Pytorch provee mayor performance y mayor flexibilidad para implementar optimizaciones en el entrenamiento. Además, posee un sintaxis simple y una asombrosa facilidad para cargar y pre-procesar imágenes.

1.1 Organización del filesystem

`ImageFolder` es la clase de Pytorch utilizada para leer, de forma simple, las imágenes desde el disco.

Para que pueda asignar las categorias a cada imagen, precisa que los datos estén organizados de forma que cada categoria tenga una carpeta y, dentro, las imágenes correspondientes.

Así que la primer tarea fue descargar las imágenes de testeo y entrenamiento, y organizarlas de forma que puedan ser leidas fácilmente por el framework. Dado que las categorias o labels se encontraban en un csv, se usaron scripts de python que organicen los archivos como se indica en la figura 1a.

El código Python utilizado para esta tarea puede ser encontrado en el repositorio en la carpeta *notebooks*.

1.2 Construcción de los conjuntos de entrenamiento, validación y testeo

En la competencia, se proveen conjuntos de entrenamiento y testeo. Decidí reservar una porción de los datos de entrenamiento como validación de forma que sirvan para observar el progreso del entrenamiento y sobre todo condiciones como Overfitting (Sobreajuste). El 20% de los datos seleccionados de manera aletoria del conjunto de entrenamiento fue usado como validación.

1.3 Data Augmentation o Image Augmentation

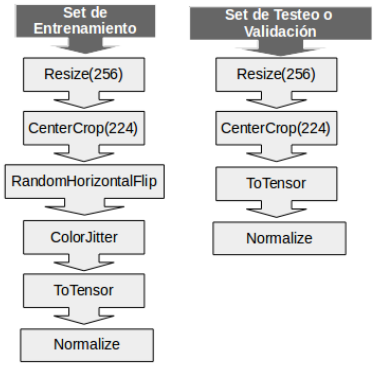
Por último, Pytorch permite construir un DataLoader que va a contener las imágenes pre-procesadas con las transformaciones que indiquemos. En el caso del dataset de entrenamiento, podemos especificar transformaciones que agreguen Image Augmentation, es decir, que apliquen rotaciones, flipeos, redimensionamientos, crops, etc para darle variabilidad a los datos.

En Pytorch, estas tranformaciones se aplican en las épocas sin agrandar el dataset pero generando diferencias en una misma imagen entre las épocas lo que permite que se puedan reajustar los pesos acorde a estas.

Hay varias tranformaciones disponibles provistas por Pytorch e incluso se pueden codear propias. En particular, trabajé con el siguiente pipeline de transformaciones:

```
data
├── test
│   └── 0
│       └── 8888.jpg
└── train
    ├── 0
    ├── 1
    ├── 10
    ├── 11
    │   └── 7473.jpg
    ├── 12
    ├── 13
    ├── 14
    │   └── 3001.jpg
    │   └── 4548.jpg
    ├── 15
    ├── 2
    ├── 3
    ├── 4
    ├── 5
    ├── 6
    ├── 7
    ├── 8
    └── 9
```

(a) Organización de directorios para Pytorch



(b) Pipeline de transformaciones

Justificación:

- Resize(256) → CenterCrop(224): Dado que el modelo va a usar Transfer Learning sobre modelos entrenados con tamaños de entrada 224x224x3, las imagenes son redimensionadas a 256x256x3 para luego, mediante CenterCrop, ignorar los bordes quedándo con el centro, donde podría estar la información más relevante.
- ToTensor: Transforma los bits en un tensor
- Normalize: Normaliza los datos para mejorar la convergencia

Para el dataset de entrenamiento, además se incluyeron las siguientes transformaciones:

- RandomHorizontalFlip: Debido al tipo de problema, una habitación vista de derecha a izquierda, o de izquierda a derecha pertenece a una misma categoría. Por ejemplo, una silla apuntando hacia la izquierda o la derecha no cambia la detección y poder invertir las imagenes permite darle mayor información al clasificador.
- ColorJitter: Aplica cambios aletorios al brillo, constranste y saturación de la imagen. La idea es despegar un poco de los colores que se ven en la habitación.

Otras transformaciones consideradas y descartadas:

- Grayscale: se probó convertir grayscale sin grandes resultados aparentemente porque los modelos pre-entrados usados trabajan con entradas a color.
- RandomRotation: Dado que las habitaciones parten de un piso, rotarlas carece de sentido. Se probó con una pequeña rotación de 10 grados sin grandes cambios.

1.4 Etiquetas y desbalanceo del conjunto de entrenamiento

El conjunto de entrenamiento suministrado está desbalanceado, es decir, algunas categorías tienen demasiadas imágenes en comparación a otras.

No modifiqué los datos ni excluí imágenes por esto. Simplemente, calculé los diferentes pesos de las categorías de las siguiente forma para ser usados por la función de pérdida (que va a ser un weighted categorical crossentropy, más información en las siguientes secciones).

2 Modelado

2.1 Transferencia del aprendizaje - Transfer Learning

En lugar de construir un modelo desde el llano, se optó por re-entrenar un modelo existente, cuyo pesos fueron calculados el dataset Imagenet.

Se elije este enforque porque estos modelos se encuentran hiper optimizados y entrenados durante días sobre los datos. Lograr algo similar requeriría mucho tiempo y esfuerzo que carecería de sentido en este contexto.

Estos modelos son, además, un buen benchmark para evaluar cualquier modelo que se quiera diseñar desde cero.

Network	Top-1 error	Top-5 error
AlexNet	43.45	20.91
VGG-11	30.98	11.37
VGG-13	30.07	10.75
VGG-16	28.41	9.62
VGG-19	27.62	9.12
VGG-11 with batch normalization	29.62	10.19
VGG-13 with batch normalization	28.45	9.63
VGG-16 with batch normalization	26.63	8.5
VGG-19 with batch normalization	25.76	8.15
ResNet-18	30.24	10.92
ResNet-34	26.7	8.58
ResNet-50	23.85	7.13
ResNet-101	22.63	6.44
ResNet-152	21.69	5.94

Network	Top-1 error	Top-5 error
SqueezeNet 1.0	41.9	19.58
SqueezeNet 1.1	41.81	19.38
Densenet-121	25.35	7.83
Densenet-169	24	7
Densenet-201	22.8	6.43
Densenet-161	22.35	6.2
Inception v3	22.55	6.44
GoogleNet	30.22	10.47
ShuffleNet V2	30.64	11.68
MobileNet V2	28.12	9.71
ResNeXt-50-32x4d	22.38	6.3
ResNeXt-101-32x8d	20.69	5.47

Figure 2: Modelos pre-entrenados - Benchmark - Fuente: Pytorch

Para la competencia, se probaron modelos simples de pocas capas, intermedios y con muchas capas. Se probó con VGG19 con BatchNormalization, Resnet18, AlexNet, Resnet50 y Resnet101. Sin embargo, aquellos que tienen más capas son los que mejores resultados tuvieron: **Resnet152** y **Denset201**.

Adicionalmente, se probó el modelo *inception* que utiliza entradas de 299x299x3. Los resultados no fueron superiores que los dos detallados anteriormente.

La elección del modelo final tuvo dos etapas. La primera relacionada con evaluar distintos modelos con diferentes learning rates, optimizadores y parámetroes, y una segunda etapa, focalizada en la optimización de los modelos de interés.

En un comienzo, no todos los entrenamientos fueron bajo las mismas condiciones, hasta que se encontraron parámetros y modelos interesantes que fueron expuestos a fine tuning que detallo en la próxima sección.

Los resultados acompañan la lógica de que modelos más profundos captan mejor los detalles de los datos pero tienden a overfittear y ese fue el principal problema que apareció durante el entrenamiento. Además, la elección de Resnet152 y Densenet201 como candidatos está también justificada por los benchmarks presentados en la figura 2.

Por falta de tiempo, excluyo de la optimización a VGG19 con Batch Normalization e Inception, que me parecen bastante interesantes y que me hubiera gustado profundizar.

2.1.1 Bottleneck features vs. entrenamiento del modelo completo

También se consideró usar los modelos como features extractors, es decir, usar los pesos sobre los datos de entrenamiento y entrenar un modelo más pequeño que utilize esta información de entrada, pero los resultados no fueron tan buenos como reentrenar el modelo completo.

La técnica de reentrenar el modelo consiste en usar los pesos pre-calculados como pesos iniciales y reentrenar el modelo completo, no únicamente las capas finales. Este segundo aproach tuvo resultados superiores.

2.2 El modelo

El modelo que mejor funcionó fue un Resnet152. Las redes neuronales residuales proponen una solución al desvanecimiento del gradiente durante el entrenamiento. Esta compuesta por bloques residuales que poseen ademas un enlace que lleva al final del bloque aplicando la función identidad sobre la entrada.

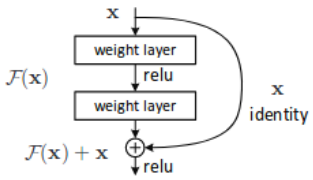


Figure 2. Residual learning: a building block.

(a) Bloque Resnet

layer_name	output_size	152-layer	
input		224x224x3	
conv1	112x122	7x7, 64, stride 2	
conv2_x	56x56	3x3 max pool, stride 2	
		1x1, 64 3x3, 64 1x1, 256	X 3
conv3_x	28x28	1x1,128 3x3,128 1x1,512	X 8
conv4_x	14x14	1x1, 256 3x3, 256 1x1, 1024	X 36
conv5_x	7x7	1x1,512 3x3, 512 1x1, 2048	X 3
output	1x1	average pull, 16 fc, softmax	

(b) Resnet-152 modificada . Salida 16 nodos.

La última capa de esta red la he reemplazado por una capa completamente conectada que retorna 16 salidas, cada una correspondiente a la probabilidad de cada categoría.

2.3 Función de pérdida

La elección de la función de pérdida (loss function) se vió relacionada con el tipo de problema a resolver. Dado que se trata de una clasificación elegí, en primer lugar, la función *Categorical Crossentropy*.

Los resultados no eran los esperados de manera equitativa en todas las categorías por la naturaleza de los datos, no se poseía una cantidad equilibrada de imágenes por categoría. Por lo tanto, decidí finalmente utlizar una variante que además considera los pesos, *Weighted Categorical Crossentropy*.

Los pesos fueron calculados de la siguiente forma:

Dado l_i con $0 \leq i \leq 15$ indicando cada una de las etiquetas:

$$weight(l_i) = count(l_{max\_j})/count(l_i)$$

siendo $0 \leq max\_j \leq 15$ y max_j id de la categoría con máxima cantidad de imágenes.

Estos pesos calculados, son pasados a la función de pérdida *Weighted Categorical Crossentropy* que los usa en el cálculo de la pérdida.

Etiqueta	Cantidad Imágenes	Pesos
Bar	164	29.7195
Bathroom	1147	4.2493
Bedroom	4874	1
Breakfast	671	7.2638
City View	529	9.2136
Dining	453	10.7594
Hotel Front	832	5.8582
Hotel Exterior	977	4.9887
Hotel Interior	794	6.1385
Kitchen	448	10.8795
Living Room	476	10.2395
Lobby	692	7.0434
Natural View	693	7.0332
Pool	426	11.4413
Recreation	255	19.1137
Sports	219	22.2557

(a) Pesos por categoria para todo el dataset

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

(b) Cálculo del Cross Entropy Loss usando pesos por Pytorch

3 Evaluación de los Resultados

3.1 Optimización

En la primer etapa de optimización busqué el modelo, optimizadores y parámetros que mejor resultados daban. Las mediciones que usé, en esta primer etapa, fueron los ploteos de accuracy y evolución de la función de pérdida, en validación y testeo.

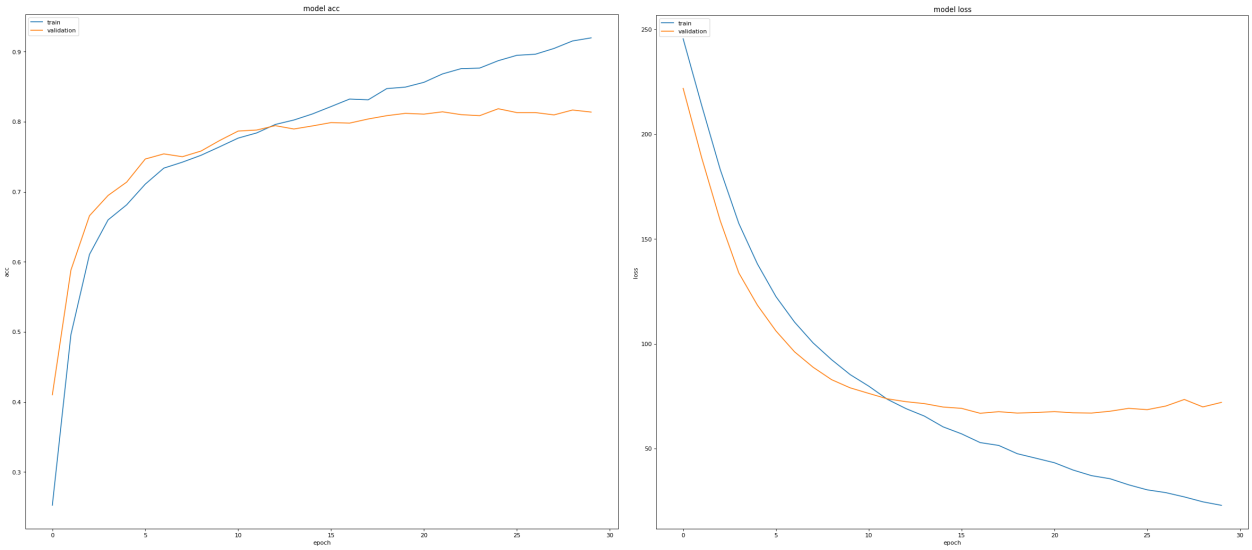
Como detallé anteriormente, los mejores resultados los obtuve con las siguiente combinaciones (en negrita):

Estrategias	Transfer Learning – **Fine Tuning** y Feature Extrator
Modelos	**VGG19 BN** – Resnet18 – Resnet50 – **Resnet152** – Densenet161 – **Densenet201** – AlexNet – **Inception**
Learning Rates	0.01, 0.001, 0.0001, 0.0001, 0.00001, 0.000001, 5e-7, 4e-7
Epochs	1-120 (**mejor rango: 10 a 30 con 1e-6**)
Batch Sizes	**16** – 32- 64
Optimizadores	**Adam** – Adagrad – **SGD** – Adamax – RMSProp
Loss Functions	Categorical Cross Entropy – **Weighted Categorical Crossentroy**
Otras	**Weight Decay – Learning Rate Decay**

Figure 5: Combinaciones y modelos testeados. En negrita, las combinaciones que tuvieron mejores resultados

Luego de la primer etapa de optimización, de la cual extraje dos modelos candidatos: Resnet153 y Densenet201, y algunos parámetros (batch_size = 16, learning_rate=1e-6 aprox.), realizé varios entrenamientos que me permitan encontrar el mejor optimizador.

El problema principal que tenía es que el entrenamiento tenía un techo del 80% de accuracy sobre el conjunto de validación. Así también la función de pérdida planchaba cuando el learning rate era constante y tendía a overfitear.



(a) Accuracy (b) Loss Plot

Figure 6: Resnet152 LR: 1e-06 30 epochs

Por lo cual, la segunda etapa, la dediqué a usar técnicas de regularización L2 como **Weight Decay** y la modificación dinámica del learning rate, **Learning Rate Decay**. Pytorch trae la posibilidad de usar diferentes Schedulers con SGD que ante determinados eventos o con diferentes técnicas reducen el Learning Rate. En particular, seleccioné probar: **Cosine** y **ReduceLROnPlateau** con diferentes parámetros, como se observa a continuación:

		Resnet152	Densenet201
SGD + Cosine	Epoch best acc val	11 de 15	12 de 15
	Val_acc	0.8161	0.8179
	Train_acc	0.8811	0.8815
	Overfit epoch	7 de 15	7 de 15
	At Overfit start Val_acc	0.8033	0.7755
	At Overfit start Train_acc	0.8167	0.8077
	Learning_rate	0.000001	1.00E-06
	Opt_params	T_max=128	T_max=128
	Score Training	0.775	0.7702
	Score Final	0.7449	0.7433
	Weigths filename	resnet152_SGD_1e-06_LRSched682_15ep16bs_1563006134.pth	densenet_SGD_1e-06_LRSched682_15ep16bs_1563037923.pth
SGD + ReduceLROnPlateau	Epoch best acc val	15 de 15	
	Val_acc	0.8201	
	Train_acc	0.9038	
	Overfit start epoch	6 de 15 (*)	
	At Overfit start Val_acc	0.7996	
	At Overfit start Train_acc	0.8025	
	Learning_rate	0.000001	
	Opt_params	threshold=1e-2, mode='min',patience=2, weight_decay=1e-2	
	Bal Acc Score Training	0.7644	
	Bal Acc Score Final	0.7499	
	Weigths filename	resnet152_SGD_1e-06_LRSched682_15ep16bs_1563143244.pth	
SGD + Cosine + weight_decay	Epoch best acc val	15 de 15	15 de 15
	Val_acc	0.8165	0.8125
	Train_acc	0.9353	0.9185
	Overfit epoch	6 de 15	8 de 15
	At Overfit start Val_acc	0.8073	0.8037
	At Overfit start Train_acc	0.7972	0.8247
	Learning_rate	1.00E-06	1.00E-06
	Opt_params	T_max=128, weight_decay=1e-2	T_max=128, weight_decay=1e-2
	Score Training	0.7632	0.7672
	Score Final	0.7499	0.7446
	Weigths filename	resnet152_SGD_1e-06_LRSched682_15ep16bs_1563086248.pth	densenet_SGD_1e-06_LRSched682_15ep16bs_1563070715.pth

Si bien Cosine tenia buen Balanced Accuracy Score sobre el dataset de entrenamiento, la realidad, es que ReduceLROnPlateu parecía lograr mayor accuracy en validación a similar Balanced Accuracy Score en entrenamiento. La métrica del accuracy sobre validación era más importante dado que el Balanced Accuracy calculado dependia de la distribución actual del entrenamiento y desconocía como sería la distribución del dataset de la segunda etapa. Por lo cual, decidí que ReduceLROnPlateu con Resnet152 era la mejor combinación sobre la cual profundizar.

Vale aclarar que usar ReduceLROnPlateu, como dice su nombre, va reduciendo el learning rate a medida que transcurre el entrenamiento observando alguna métrica que uno puede indicar. En particular, elegí observar el loss en validación. Cuando se llegaba a un estancamiento, el scheduler lo reducía automáticamente. La idea era evitar el planchado mostrado en los gráficos anteriores. Incluso se puede elegir ser más agresivo con la reducción, pero por cuestiones de tiempo, no pude experimentar mucho con ese parámetro.

Así la siguiente optimización estuvo dada por el siguiente cuadro:

Optimizador	Tecnica LR	LR	Weight_decay	Opt_params	Epochs	Epoch best	Train_acc	Val_acc	Bal Acc Score Training	Bal Acc Score Final
SGD	ReduceLROnPlateau	1.00E-04	0	threshold=1e-2 mode='min' patience=0 eps=1e-10	15	13	0.9015	0.8099	0.7613	
SGD	ReduceLROnPlateau	1.00E-04	0.1	threshold=1e-2 mode='min' patience=0 eps=1e-10	15	13	0.9013	0.815	0.7778	0.7603
SGD	ReduceLROnPlateau	1.00E-04	0.01	threshold=1e-2 mode='min' patience=0 eps=1e-10	15	7	0.9025	0.8106	0.7691	0.7578
SGD	ReduceLROnPlateau	1.00E-05	0.01	threshold=1e-2 mode='min' patience=0 eps=1e-10	10	10	0.8305	0.8062	0.7699	0.7487
SGD	ReduceLROnPlateau	1.00E-06	0.02	threshold=1e-2 mode='min' patience=0 eps=1e-10	15				0.6898	
SGD	ReduceLROnPlateau	1.00E-05	0.1	threshold=1e-2 mode='min' patience=0 eps=1e-10	10	8	0.8355	0.8106	0.7818	0.7595

El indicado como test1 es la combinación que pasó a la segunda etapa. Lamentablemente, no llegué a subir la combinación test6 que se veía más balanceada en accuracy (overfiteaba menos) logrando casi los mismos scores.

3.2 Visualización de los datos y las soluciones

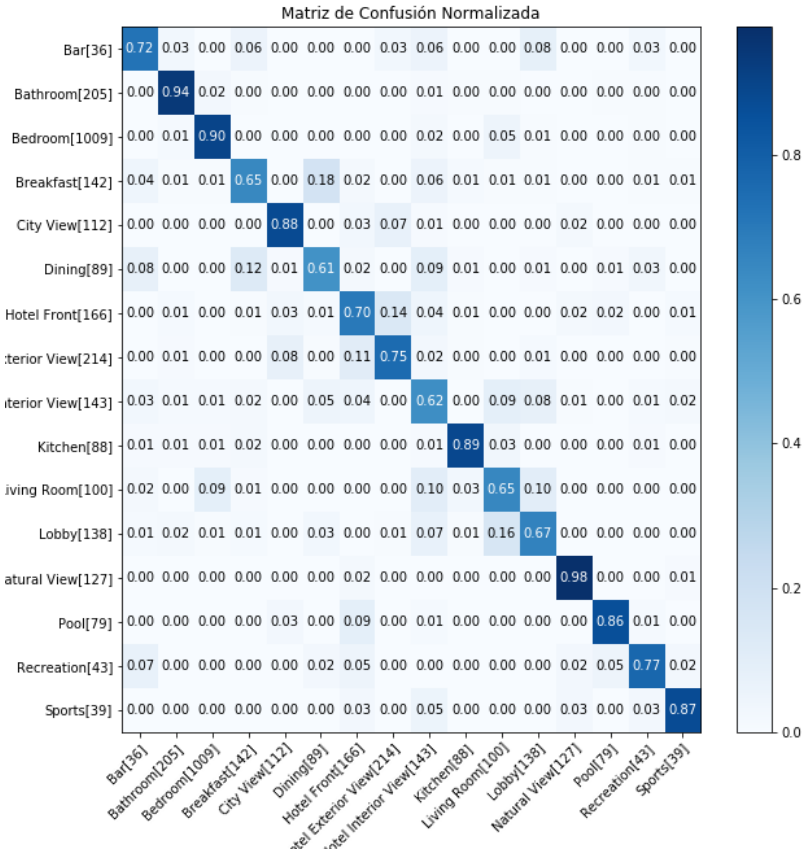
La solución final estaría dada por el modelo, función de pérdida, optimizador y parámetros:

Modelo: Resnet152 + FC 16 softmax (Transfer Learning - Fine Tuning/Reentrenar con pesos)

Función de Pérdida: Weighted Categorical Crossentropy (torch.nn.CrossEntropyLoss weight=weights)

Optimizador: SGD (learning_rate = 1e-4, weight_decay=0.1)

SGD Scheduler: ReduceLROnPlateau (threshold=1e-2 mode='min',patience=0,eps=1e-10) sobre el loss en validación
Batch Size: 16
Epochs: 13



Se deduce que por ejemplo, confunde el 18% de los Breakfast con Dinning, lo cual suena lógico dado que ambos espacios se usan para comer y por ejemplo, pueden contar con mesas y sillas.
Luego, confunde un 16% los Living Room con Lobby, similar razonamiento, ambos podrían incluir sillones.

(a) Dinning (b) Breakfast

Figure 7: Errores de clasificación