Aprendizaje Profundo para Procesamiento de Señales de Imagen y Vídeo
Práctica 1

Antonio Coín Castro

21 de abril de 2021

1. SimpleCNN

En el primer notebook, `Image_Classification_1.ipynb`, desarrollamos una arquitectura CNN simple para clasificación de imágenes del conjunto MNIST (dígitos manuscritos en escala de grises).

Resumen del dataset

En `Pytorch` disponemos directamente del dataset MNIST como *built-in*.

	Alto de imagen	Ancho de imagen	Nº canales de imagen	Nº muestras
Entrenamiento	28	28	1	60000
Validación	28	28	1	10000

Arquitectura de SimpleCNN

Construimos una red sencilla con dos capas convolucionales con kernels 3×3 y activación ReLu, y dos capas totalmente conectadas. Añadimos una capa *Max-Pooling* 2×2 para realizar *downsampling* y capas *Dropout* para añadir regularización. En resumen, la arquitectura queda como sigue:

Capa	Output shape	Nº canales output	Nº parámetros entrenables
Conv2D	28×28	32	320
ReLu	28×28	32	0
Conv2D	28×28	32	9248
ReLu	28×28	32	0
MaxPool2D	14×14	32	0
Dropout(0.25)	14×14	32	0
Flatten	6272×1	1	0
Linear	128×1	1	802944
ReLu	128×1	1	0
Dropout(0.5)	128×1	1	0
Linear	10×1	1	1290
Nº total de parámetros			813802

Curvas de aprendizaje y precisión

Entrenamos nuestro modelo durante 10 épocas, obteniendo las curvas de entrenamiento y validación de la Figura 1.

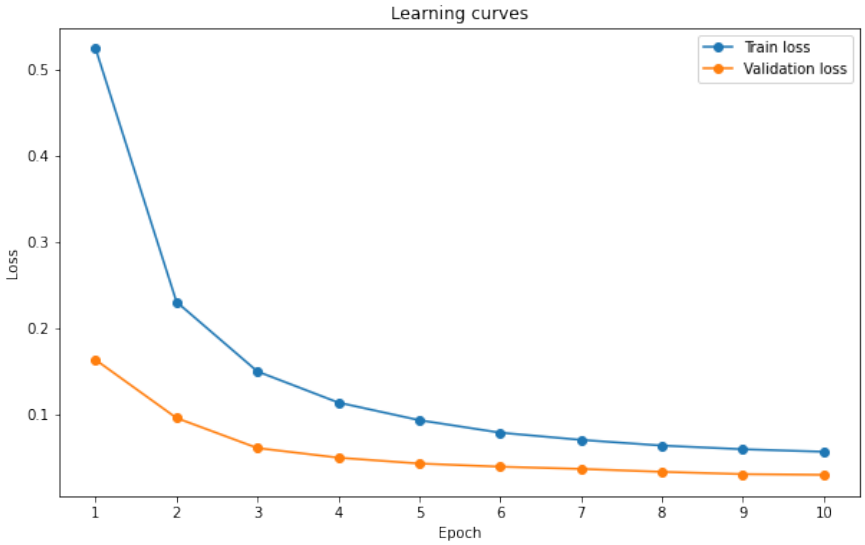


Figura 1: Curvas de entrenamiento y validación de SimpleCNN en 10 épocas.

Observamos que no se aprecia *overfitting*, ya que la pérdida en validación se mantiene siempre por debajo de la pérdida en entrenamiento, y además ambas curvas van descendiendo conforme aumentan las épocas. Tampoco se aprecian problemas de *underfitting* ó *high-bias*, ya que el modelo sí que está aprendiendo (como hemos dicho, la pérdida va disminuyendo). Esto se debe muy probablemente, además de a la adecuación del modelo a los datos, a la presencia de capas de regularización *Dropout* en entrenamiento (y obviamente su ausencia en validación), que hacen que aumente la capacidad de generalización.

En cuanto a la precisión en validación, obtenemos el siguiente resultado:

	Mejor precisión (validación)	Época con mejor precisión
SimpleCNN	$98.98\,\%$	10

Vemos que es un muy buen resultado, con una precisión bastante cercana al $100\,\%$. Vistas las curvas de aprendizaje, es probable que el modelo siguiera aprendiendo si seguimos entrenando durante más épocas, ya que parece que la pérdida en validación sigue descendiendo y no ha llegado aún al punto óptimo. De hecho, podemos comprobar esta hipótesis continuando con el entrenamiento del modelo durante 10 épocas más. Si lo hacemos, obtenemos la curva completa de aprendizaje de la Figura 2. Efectivamente, conseguimos aumentar la precisión en validación máxima hasta el $99.22\,\%$, y ya la curva de validación parece haberse estancado e incluso empieza a aumentar ligeramente, por lo que el modelo probablemente ya no se beneficiaría mucho de más épocas de entrenamiento.

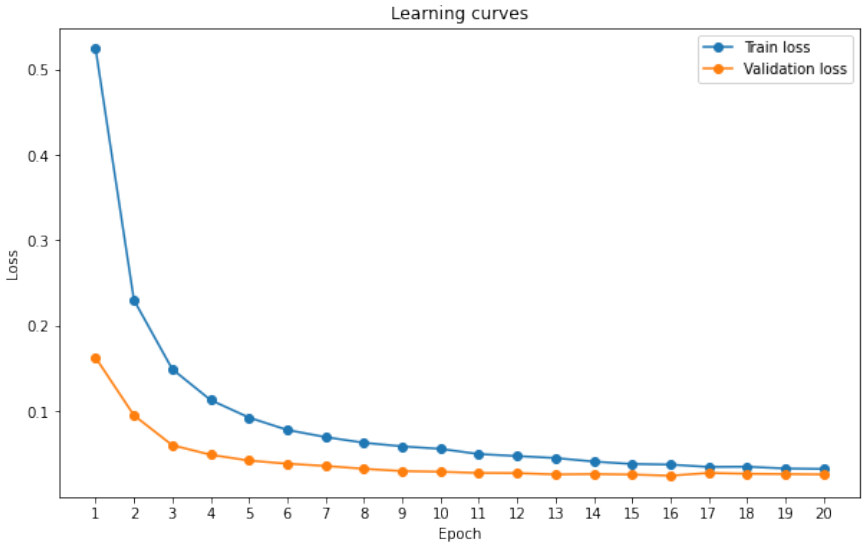


Figura 2: Curvas de entrenamiento y validación de SimpleCNN en 20 épocas.

Matriz de confusión

Otra de las formas de evaluar el modelo es mostrar la matriz de confusión, que mide el número de aciertos y fallos entre las predicciones y los valores reales para todas las etiquetas por separado. En nuestro caso podemos verla para el conjunto de validación en la Figura 3.

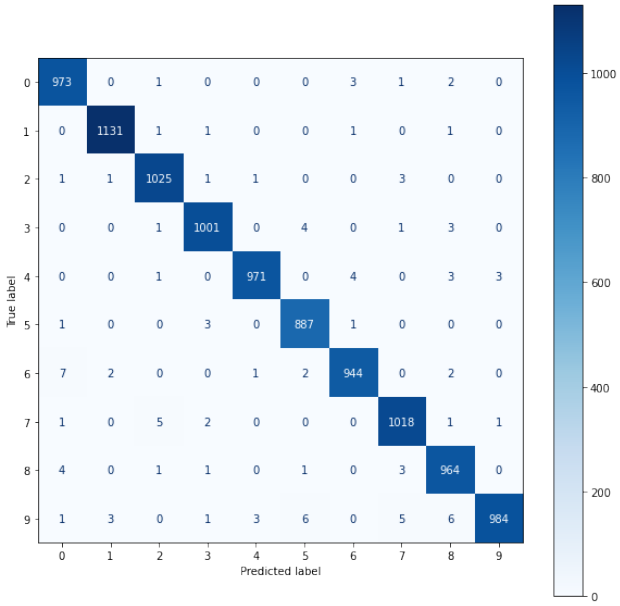
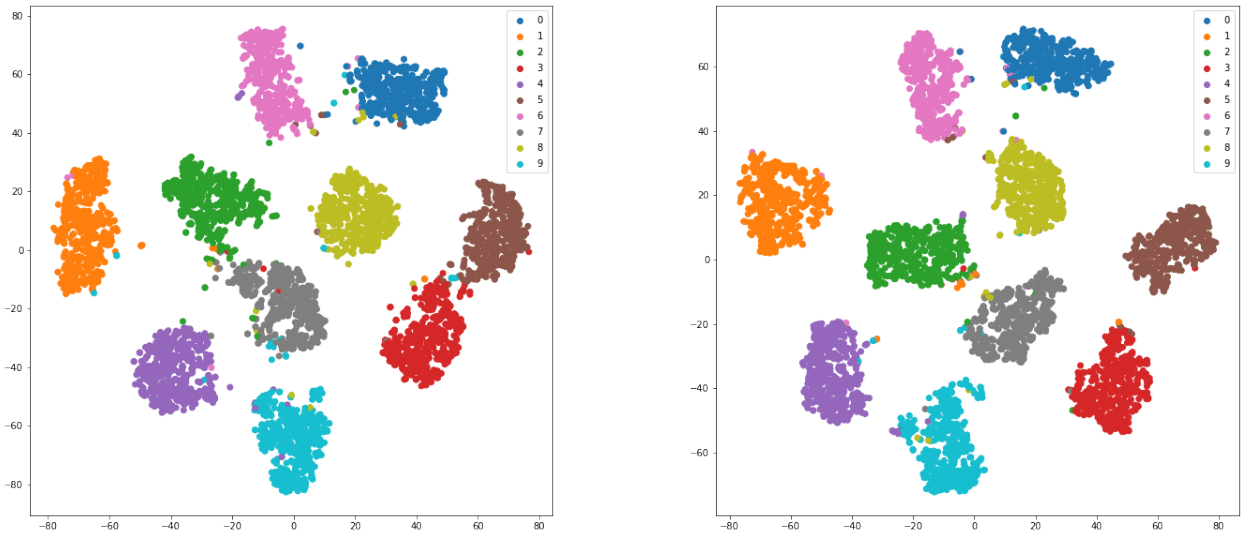


Figura 3: Matriz de confusión en validación para el modelo SimpleCNN tras 10 épocas.

Analizando los resultados, vemos que en total cometemos 102 errores (de un total de 10000 predicciones). Los dígitos que más se confunden son el 6 con el 0, con 7 fallos, seguidos del 9 con el 5 y el 8, con 6 fallos en cada caso. Todos ellos son dígitos relativamente fácilmente de confundir por su forma, dependiendo de la persona que los escriba.

Proyecciones t-SNE

Incluimos a continuación en la Figura 4 las proyecciones t-SNE de la representación de las capas intermedia y final de la red para los datos de validación.



(a) Representación intermedia

(b) Representación final

Figura 4: Proyecciones t-SNE de las representaciones de SimpleCNN.

Como vemos, con la representación intermedia ya obtenemos 10 clústers claramente diferenciados, cada uno correspondiente a un dígito distinto. El hecho de que se formen estos grupos tras la proyección en 2D hace pensar que la capacidad discriminativa de la red tras la capa intermedia es suficiente para clasificar con éxito los dígitos. Sin embargo, vemos que se observa cierto solapamiento en algunos clústers (por ejemplo en el 7 y en el 2), y que otros grupos están un poco dispersos (por ejemplo el 1). Si pasamos a la representación final vemos que los clústers están ya más separados y más compactos, lo que sugiere que la clasificación a partir de las características finales será aún mejor[1].

[1] Aunque tras la representación t-SNE obtengamos clústers diferenciados, esto no garantiza en principio que la clasificación en el espacio original vaya a ser igual de buena, aunque es probable que así sea.

2. AlexNet

En este caso pasamos al conjunto CIFAR10, también disponible en `Pytorch`, que consta en total de 60000 imágenes RGB pertenecientes a diez clases balanceadas. El notebook asociado es `Image_Classification_2.ipynb`.

Arquitectura de AlexNet

```
class AlexNet(nn.Module):
  def __init__(self, output_dim):
    super(AlexNet, self).__init__()

    self.features = nn.Sequential(
      nn.Conv2d(3, 48, 5, 2, 2),
      nn.MaxPool2d(2),
      nn.ReLU(inplace = True),

      nn.Conv2d(48, 128, 5, 1, 2),
      nn.MaxPool2d(2),
      nn.ReLU(inplace = True),

      nn.Conv2d(128, 192, 3, 1, 1),
      nn.ReLU(inplace = True),
      nn.Conv2d(192, 192, 3, 1, 1),
      nn.ReLU(inplace = True),

      nn.Conv2d(192, 128, 3, 1, 1),
      nn.MaxPool2d(2),
      nn.ReLU(inplace = True)
  )

  self.classifier = nn.Sequential(
      nn.Dropout(0.5),
      nn.Linear(128*2*2, 2048), # final conv layer resolution 2x2
      nn.ReLU(inplace = True),

      nn.Dropout(0.5),
      nn.Linear(2048, 2048),
      nn.ReLU(inplace = True),

      nn.Linear(2048, output_dim)
    )

  def forward(self, x):
      x = self.features(x)
      interm_features = x.view(x.shape[0], -1)
      x = self.classifier(interm_features)
      return x, interm_features
```

Podemos ver a continuación el modelo generado a partir del código anterior, donde apreciamos que el número total de parámetros entrenables asciende a 6199498.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 48, 16, 16]           3,648
         MaxPool2d-2            [-1, 48, 8, 8]               0
              ReLU-3            [-1, 48, 8, 8]               0
            Conv2d-4           [-1, 128, 8, 8]         153,728
         MaxPool2d-5           [-1, 128, 4, 4]               0
              ReLU-6           [-1, 128, 4, 4]               0
            Conv2d-7           [-1, 192, 4, 4]         221,376
              ReLU-8           [-1, 192, 4, 4]               0
            Conv2d-9           [-1, 192, 4, 4]         331,968
             ReLU-10           [-1, 192, 4, 4]               0
           Conv2d-11           [-1, 128, 4, 4]         221,312
        MaxPool2d-12           [-1, 128, 2, 2]               0
             ReLU-13           [-1, 128, 2, 2]               0
          Dropout-14                 [-1, 512]               0
           Linear-15                [-1, 2048]       1,050,624
             ReLU-16                [-1, 2048]               0
          Dropout-17                [-1, 2048]               0
           Linear-18                [-1, 2048]       4,196,352
             ReLU-19                [-1, 2048]               0
           Linear-20                  [-1, 10]          20,490
================================================================
Total params: 6,199,498
Trainable params: 6,199,498
Non-trainable params: 0
----------------------------------------------------------------
```

Curvas de aprendizaje y precisión

Si entrenamos el modelo durante 15 épocas, obtenemos las curvas de entrenamiento y validación de la Figura 5. Vemos que en un principio el modelo aprende sin caer en *overfitting* (ambas pérdidas van descendiendo casi a la par), pero a partir de aproximadamente la época 10 la pérdida en validación comienza a estancarse y luego a aumentar. Esto nos lleva a la conclusión de que hemos pasado el punto óptimo de *early stopping* y el modelo está empezando a sobreajustar, y que por tanto continuar con más épocas no mejoraría el rendimiento del mismo.

Por otra parte, los mejores resultados de precisión en validación son los siguientes:

	Mejor precisión (validación)	Época con mejor precisión
AlexNet	70.65 %	14

En este caso el conjunto de datos es más difícil de aprender, pero con un modelo relativamente simple como AlexNet obtenemos una precisión en torno al 70 % en unas pocas épocas.

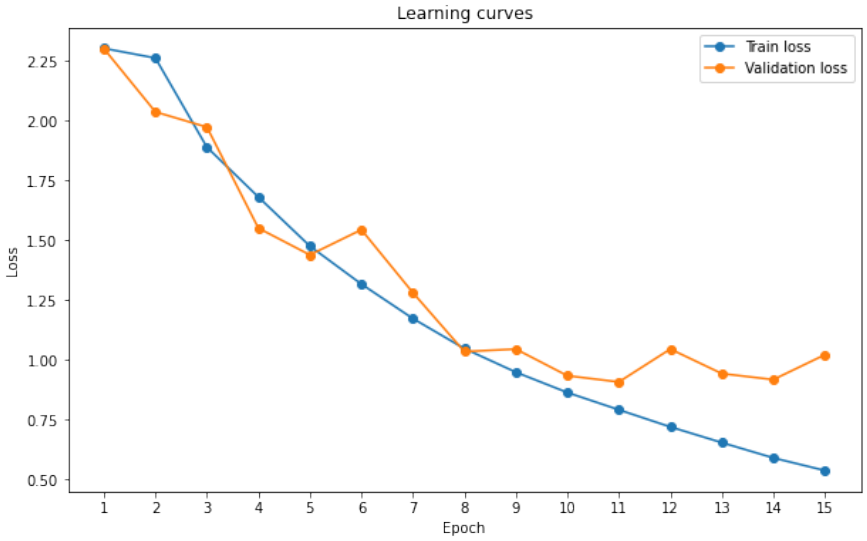


Figura 5: Curvas de entrenamiento y validación de AlexNet en 15 épocas.

Matriz de confusión

La matriz de confusión para el conjunto de validación podemos verla en la Figura 6.

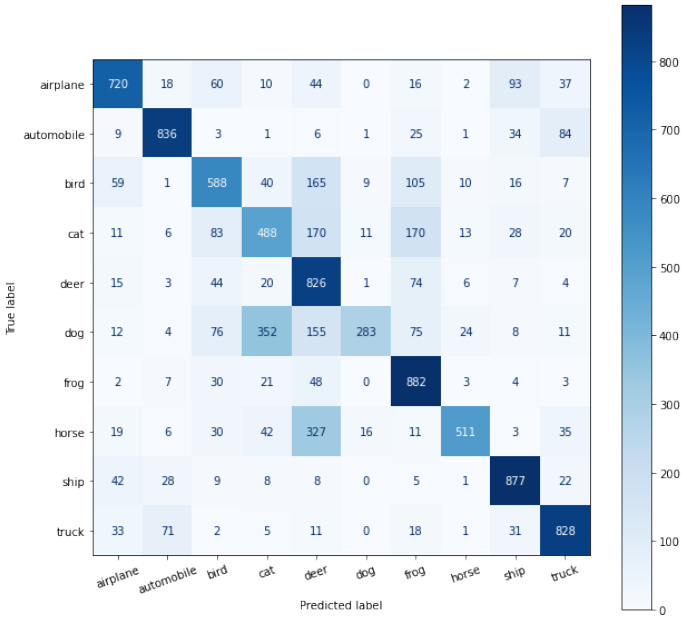


Figura 6: Matriz de confusión en validación para el modelo AlexNet tras 15 épocas.

Vemos que las clases que más se confunden son *'dog'* por *'cat'* y *'horse'* por *'deer'*. Esto tiene sentido, ya que en ambos casos se trata de animales de tamaño, color y características morfológicas parecidas, que dependiendo de la distancia, pose u oclusión pueden ser difíciles de diferenciar incluso para los seres humanos. También hay que tener en cuenta que es muy probable que las imágenes de las clases confundidas estén en un contexto parecido (ambientes domésticos para perros y gatos, y naturaleza para caballos y ciervos), dificultando aún más la clasificación.

Proyección t-SNE

Podemos mostrar aquí también la proyección t-SNE de la representación de la capa final de la red, que se observa en la Figura 7.

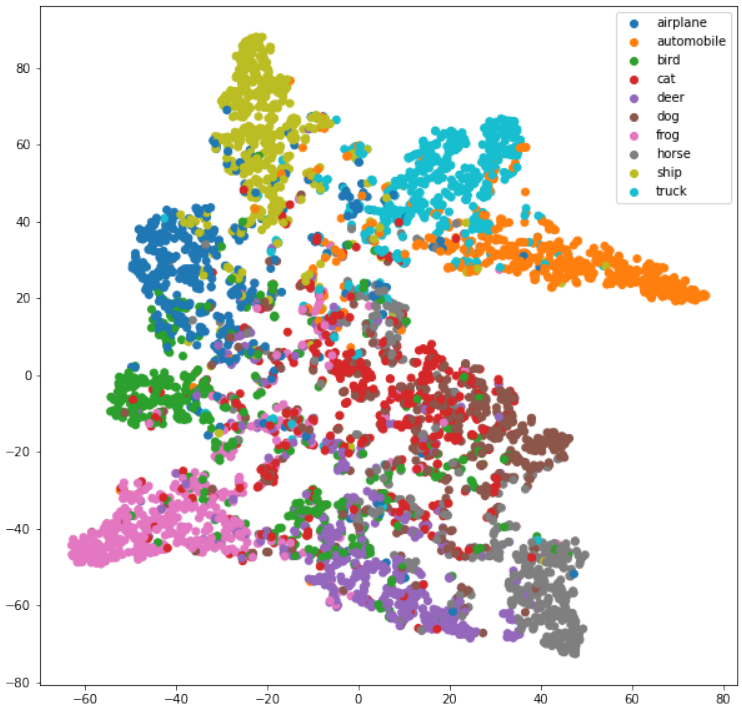


Figura 7: Proyección t-SNE en validación con la representación final de AlexNet.

En este caso no se observan clústers tan diferenciados y compactos como en MNIST. Sin embargo, vemos que las diferentes clases sí forman grupos más o menos separados, aunque con bastante dispersión y solapamiento. Por ejemplo, vemos que los clústers correspondientes a perros y gatos son los más dispersos y solapados, pues como dijimos antes estas clases son las dos que más se confunden. Algo parecido ocurre con las agrupaciones asociadas a pájaros y aviones o a camiones y coches, que también son confundidas con frecuencia por el modelo debido a las características de las imágenes: forma, tamaño, *background* común, etc. Por todo esto, podemos pensar que la clasificación no será tan buena como en MNIST (de hecho, hemos comprobado que no lo es), pues el modelo no posee suficiente capacidad de discriminación.

Podríamos intentar aumentar el rendimiento del modelo con técnicas como *data augmentation*, y también optimizando los hiperparámetros del modelo: el tamaño de *batch*, el *learning rate*, el propio optimizador, etc.

3. Transfer learning

Esta última parte corresponde al notebook `Transfer_Learning.ipynb`, donde experimentamos con técnicas de *transfer learning* y *finetuning* en el conjunto de datos externo Alien vs. Predator, con un total de 894 imágenes RGB pertenecientes a dos clases. Empleamos cuatro enfoques distintos para intentar clasificar las imágenes:

1. Entrenar un modelo desde cero (*Scratch*).
2. Utilizar una red preentrenada como extractor de características y clasificar con SVM lineal (*Extractor + SVM*).
3. Utilizar una red preentrenada con la última capa totalmente conectada adaptada a nuestro problema, y entrenarla unas pocas épocas más mediante finetuning (*Finetune*).
4. Igual que el anterior, pero haciendo *data augmentation* (*Finetune + augmentation*).

Rendimiento de los modelos

A continuación se muestra la máxima precisión conseguida con cada uno de los cuatro métodos:

	Scratch	Extractor + SVM	Finetune	Finetune + augmentation
Precisión máxima	73.50 %	90.50 %	93.00 %	95.50 %

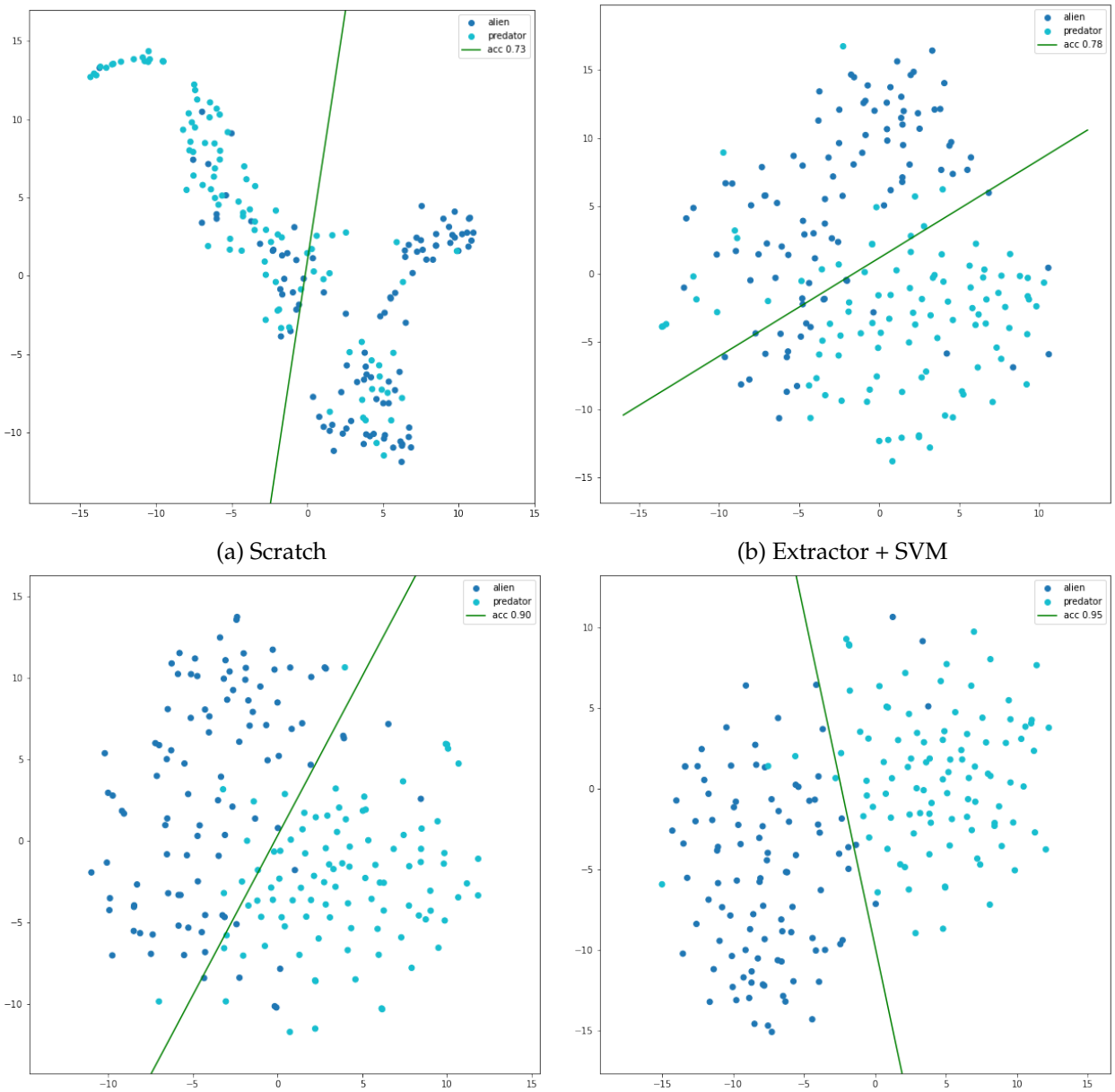
Vemos que hay una diferencia notable entre partir de un modelo sin entrenar y utilizar un modelo preentrenado, siendo esta última opción la que mejores resultados arroja.

Proyecciones t-SNE

Finalmente, mostramos en la Figura 8 las proyecciones t-SNE obtenidas a partir de las representaciones finales de cada uno de los cuatro métodos. Junto a estas proyecciones mostramos también la frontera del hiperplano que "mejor" separa linealmente los datos proyectados (según un SVM lineal entrenado con ellos).

Observamos que en el caso de entrenamiento desde cero apenas se diferencian los clústers en la proyección, ya que están muy dispersos y solapados. En el resto de alternativas sí que vemos un patrón más claro en la nube de puntos, mejorando progresivamente y llegando a una separación casi perfecta con la última técnica de finetuning con data augmentation.

En cuanto al grado de separabilidad lineal, el método 1 basado en entrenamiento desde cero presenta unos resultados que dejan bastante que desear, ya que la representación obtenida no es muy buena. Por otra parte, en el método 2 de extracción de características la proyección tiene un grado aceptable de separabilidad lineal, aunque lejos de ser perfecto. El método 3 mejora bastante la representación y la separabilidad, aunque aún comete algunos fallos. Finalmente, en el caso 4 de finetuning con data augmentation obtenemos a la vez una separación de los datos proyectados y una precisión de clasificación en el espacio original casi perfectas, por lo que es claramente el modelo ganador. No obstante, para otros datasets o si dispusiéramos de menos datos de entrenamiento podría ser que algún otro de los enfoques fuera más adecuado.



(a) Scratch

(b) Extractor + SVM

(c) Finetune

(d) Finetune + augmentation

Figura 8: Proyecciones t-SNE de las representaciones finales obtenidas en validación junto a los hiperplanos separadores para cada uno de los métodos probados.