Doble Grado en Ingeniería Informática y Matemáticas

VISIÓN POR COMPUTADOR
(E. Computación y Sistemas Inteligentes)

TRABAJO-2: Redes neuronales convolucionales

UNIVERSIDAD DE GRANADA

Carlos Santiago Sánchez Muñoz

Grupo de prácticas 2 - Jueves

Email: carlossamu7@correo.ugr.es

20 de noviembre de 2019

Índice

Apartado 1

BaseNet en CIFAR100

Vamos a estudiar el entrenamiento de redes neuronales convolucionales profundas, usando Keras. Para ello empezamos trabajando con una parte del conjunto de datos `CIFAR100`.

No vamos a usar todo este conjunto de datos, nos vamos a quedar con un subconjunto de 25 clases. El conjunto de entrenamiento tiene 12500 imágenes y el de prueba 2500 imágenes que son en color de 3 canales de 32x32 píxeles. Del conjunto de entrenamiento se reservará un 10 % para validación.

Procedemos a crear el modelo `BaseNet` pedido. Observemos el código:

```
def model_baseNet(input_shape=(32,32,3)):
  model = Sequential()
  model.add(Conv2D(6, kernel_size=(5,5), activation='relu', input_shape=input_shape))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Conv2D(16, kernel_size=(5,5), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Flatten())
  model.add(Dense(50, activation='relu'))
  model.add(Dense(25, activation='softmax'))
  return model
```

Layer No.	Layer Type	Kernel size (for conv layers)	Input | Output dimension	Input | Output channels (for conv layers)
1	Conv2D	5	32 | 28	3 | 6
2	Relu	-	28 | 28	-
3	MaxPooling2D	2	28 | 14	-
4	Conv2D	5	14 | 10	6 | 16
5	Relu	-	10 | 10	-
6	MaxPooling2D	2	10 | 5	-
7	Linear	-	400 | 50	-
8	Relu	-	50 | 50	-
9	Linear	-	50 | 25	-

Ahora tenemos que definir el optimizador y compilador del modelo. Tenemos dos opciones:

```
optimizer = optimizer=keras.optimizers.Adadelta()
optimizer = SGD (lr = 0.01, decay = 1e-6, momentum = 0.9, nesterov = True)
```

El primer optimizador es el `Adadelta` que es el que se usaba por ejemplo en `mnist`. La otra opción es usar gradiente descendente estocástico. Compilamos nuestro modelo:

```
def optimizadorCompilador(model):
  model.compile(loss=keras.losses.categorical_crossentropy,
            optimizer=keras.optimizers.Adadelta(),
            metrics=['accuracy'])

  weights = model.get_weights()
  return weights
```

Es importante que teniendo el modelo base guardemos los pesos aleatorios con los que empieza la red, para poder reestablecerlos después y comparar resultados entre no usar mejoras y sí usarlas. Definimos la función pérdida a minimizar. Como hacemos clasificación multiclase usamos `categorical_crossentropy`. El argumento `metrics` son las métricas que se quieren calcular a lo largo de todas las épocas.

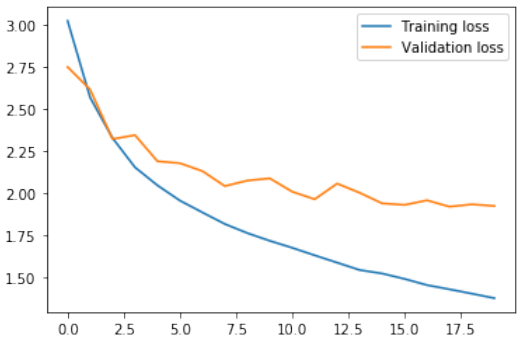
Ahora definimos el entrenamiento del modelo. Usamos la función `fit_generator()` aunque también existe la posibilidad de usar `fit()`. Tal y como hemos dicho antes cuando vamos a hacer varias llamadas a estas dos funciones sobre el mismo modelo definido previamente, con distintos argumentos en ImageDataGenerator tenemos que reestablecer los pesos de la red a como estaban antes del entrenamiento.

```
def train(model, x_train, y_train, datagen, batch_size=64, epochs=20, verbose=0):
  train_data = datagen.flow(x_train, y_train, batch_size=batch_size, subset='training')
  validation_data = datagen.flow(x_train, y_train, batch_size=batch_size, subset='validation')

  hist = model.fit_generator(train_data,
            steps_per_epoch = len(x_train)*0.9/batch_size,
            epochs = epochs,
            verbose = verbose,
            validation_data = validation_data,
            validation_steps = len (x_train)*0.1/batch_size)

  return hist
```

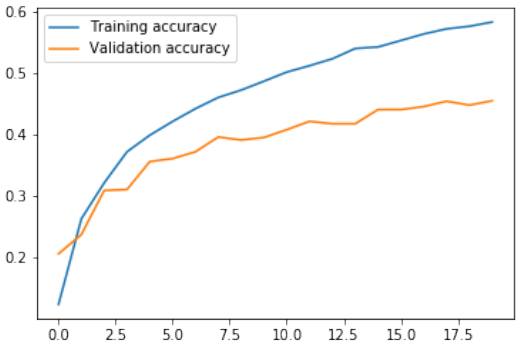
Necesitamos ahora una función que ejecute todo lo que nos han pedido en este apartado y observar el rendimiento de BaseNet. Para ello he creado la función `ejercicio1` que carga las imágenes, crea el modelo y lo compila y por último entrena y nos da una predicción acerca del `text`.

```
def ejercicio1():
  input_shape=(32, 32, 3)       # Imágenes en color con 3 canales de 32x32 píxeles.
  num_classes = 25              # Número de clases es 25
  batch_size = 64               # Tamaño de batch potencia de 2
  epochs = 20                   # Elegimos número de épocas

  x_train, y_train, x_test, y_test = cargarImagenes()
  datagen = ImageDataGenerator(validation_split = 0.1)
  datagen.fit(x_train)

  model = model_baseNet(input_shape)
  model.summary()               # Descripción del modelo
  weights = optimizadorCompilador(model)
  model.set_weights(weights)    # Reestablecemos los pesos

  hist = train(model, x_train, y_train, datagen, batch_size, epochs, verbose=1)
  print(hist)
  mostrarEvolucion(hist)
  prediccion(model, x_test, y_test)
```

Obtenemos los siguientes resultados.



Pérdida BaseNet con opt. Adadelta

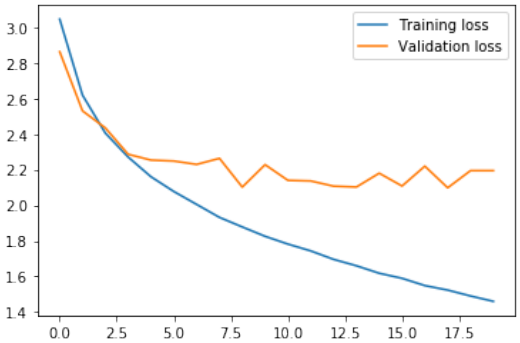


Acierto BaseNet con opt. Adadelta

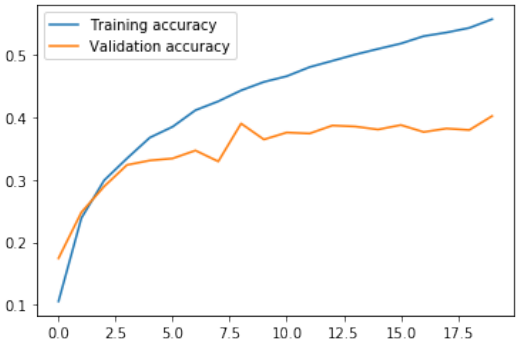
```
Test loss: 1.895128540802002
Test accuracy: 0.4632
```

Observamos diferentes cosas con una importancia llamativa. En primer lugar, tanto acierto como pérdida del conjunto de entrenamiento se alejan de la del conjunto de validación. Esto se debe a que hay un pequeño *overfitting* o sobreajuste. La red no sólo aprende datos generales y de la distribución de los datos de entrenamiento sino que también aprende las particularidades de esos datos como si los estudiase y por eso al probar con los de validación se produce esto.

En segundo lugar decir que empecé con un número de épocas de 10 y lo aumenté a 20 porque aún podía mejorar más, incluso ahora también puede pero no merece la pena porque vamos a mejorar nuestro modelo en el siguiente apartado. Destacar que la tasa de *accuracy* es bastante alta ya que tras la mejora se nos pide acercarnos al 50 % y casi lo tenemos ya.



Pérdida BaseNet con opt. SGD



Acierto BaseNet con opt. SGD

```
Test loss: 2.1183154239654542
Test accuracy: 0.4144
```

Con el optimizador `SGD` (Gradiente Descendente Estocástico) los resultados han empeorado y también observamos un poco de *overfitting*.

Apartado 2

Mejora del modelo

Nuestro objetivo es crear una red profunda mejorada haciendo elecciones juiciosas de arquitectura e implementación. Se nos pide que nuestra precisión se acerque al 50 %.

Para mejorar la red vamos a considerar los siguientes aspectos que se han ido implementando de forma incremental por lo que el último tiene los cambios y mejoras de todos los anteriores.

2.1 Normalización de datos

Vamos a normalizar los datos de entrada con el objetivo de que el entrenamiento sea más fácil y más sólido. Utilizamos la clase `ImageDataGenerator` con los parámetros correctos para que los datos estén bien condicionados (media=0, std dev=1) para mejorar el entrenamiento. Los parámetros para este propósito son:

```
featurewise$_center=True, featurewise_std_normalization=True
```

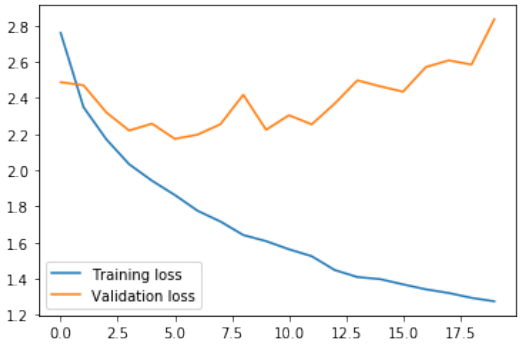
El primer parámetro establece la media de los datos a 0 y el segundo divide las entradas por su desviación típica. Con lo anterior normalizamos el conjunto de entrenamiento pero no se nos puede olvidar normalizar el conjunto de validación.

Como estamos usando `ImageDataGenerator` usamos la función `fit_generator()` en vez de `fit()`. De forma análoga a esto tenemos la opción de usar `predict()` y `predict_generator()` en la predicción y nosotros vamos a usar la segunda estableciendo el `batch_size=1` y `shuffle=False`. Por último para sacar el *accuracy* usamos la función que nos dan `calcularAccuracy()`.

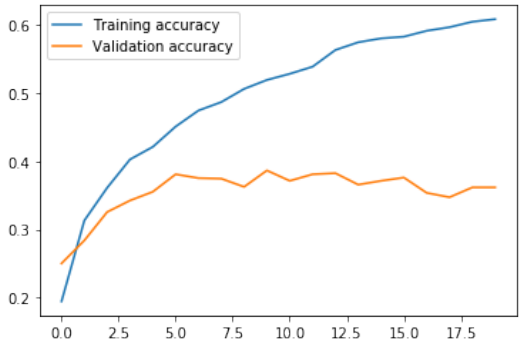
```
def ejercicio2_normalizacion():
  input_shape=(32, 32, 3)      # Imágenes en color con 3 canales de 32x32 píxeles.
  num_classes = 25             # Número de clases es 25
  batch_size = 64              # Tamaño de batch potencia de 2
  epochs = 20                  # Elegimos número de épocas

  x_train, y_train, x_test, y_test = cargarImagenes()
  datagen_train = ImageDataGenerator(featurewise_center=True,
        featurewise_std_normalization=True, validation_split=0.1)
  datagen_test = ImageDataGenerator(featurewise_center=True,
        featurewise_std_normalization=True)
  datagen_train.fit(x_train)
  datagen_test.fit(x_train)

  model = model_baseNet(input_shape)
  model.summary()              # Descripción del modelo
  weights = optimizadorCompilador(model)
  model.set_weights(weights)   # Reestablecemos los pesos

  hist = train(model, x_train, y_train, datagen_train, batch_size, epochs, verbose=1)
  mostrarEvolucion(hist)
  preds = model.predict_generator(datagen_test.flow(x_test, batch_size = 1, shuffle = False),
```

```
        steps = len(x_test))
score = calcularAccuracy(y_test, preds)
print('Test accuracy:', score)
```

En el ejercicio 2 comienzo a usar SGD como optimizador para ir haciendo mejoras.



Pérdida de BaseNet normalizado



Acierto BaseNet normalizado

2.2 Aumento de datos

Para el aumento de datos hay que entender que lo que metemos tiene que tener la misma distribución que las imágenes que la red está aprendiendo. En `mnist` los números van de 0 a 9 si hay un 5 no puedo meter un 5 tras hacerle `vertical_flip` porque no sería un 5, sería como un 2. Lo que sí puede ser interesante es rotarlo. Pongo un número bajo para que el 5 esté en la imagen. Por tanto los parámetros a usar dependen del conjunto de datos.

En el siguiente link he encontrado diferentes cambios que se pueden hacer para aumentar nuestro conjunto de datos:

`https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/`

Los cambios mas interesantes son:

i) Cambio de imagen mediante `width_shift_range` and height_shift_range arguments.

ii) Hacer flip a la imagen a través de `horizontal_flip` y `vertical_flip`.

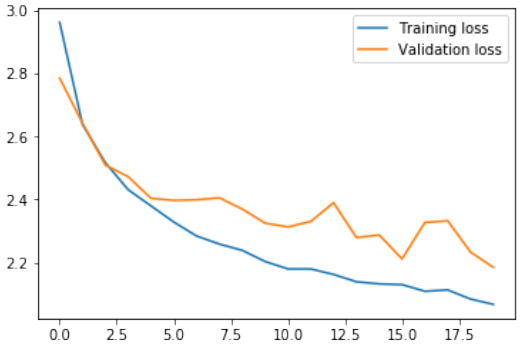
iii) Rotar la imagen mediante `rotation_range`.

iv) Cambiando el brillo de la imagen brightness mediante el parámetro `brightness_range`.

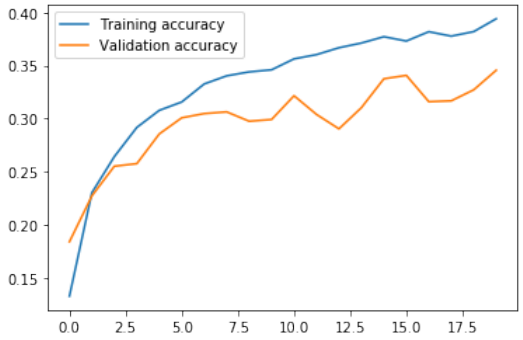
v) Haciendo zoom mediante el parámetro `zoom_range`.

He realizados dos versiones de aumento de datos, una básica usando la idea del enunciado en el que hacemos `horizontal_flip=True` y `random_crop=0.7` lo cual es coherente con nuestro sistema (hacer flip horizontal de un pájaro nos da otro pájaro).

```
def ejercicio1():
```



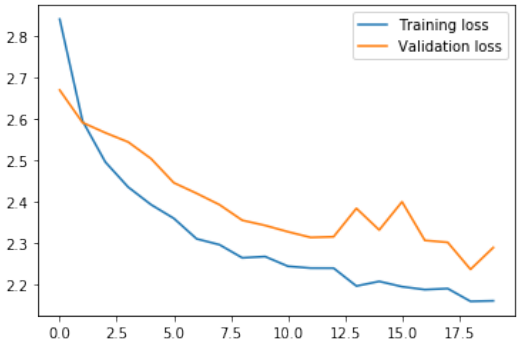
Pérdida de BaseNet con `data_augmentation`



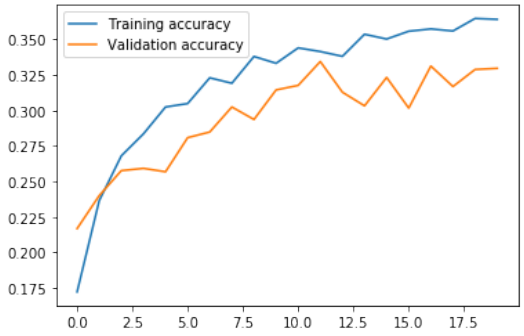
Precisión de BaseNet con `data_augmentation`

He probado a hacer cambios de brillo en el aumento de datos pero el resultado ha sido malo. Luego he hecho otro aumento de datos en el que he incluido pequeñas rotaciones (he pasado como argumento 10 grados) más lo que ya teníamos y el resultado es bueno pero el anterior era incluso mejor, veamos:

```
ImageDataGenerator(featurewise_center=True,
                   featurewise_std_normalization=True,
                   horizontal_flip=True,
                   brightness_range=[0.2,1.0],
                   zoom_range=0.7,
                   rotation_range=10,
                   validation_split=0.1)
```

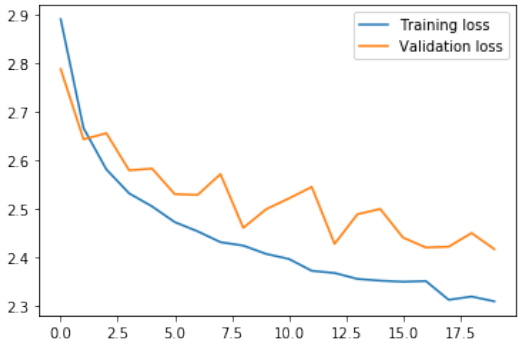


Pérdida de BaseNet con `data_augmentation`

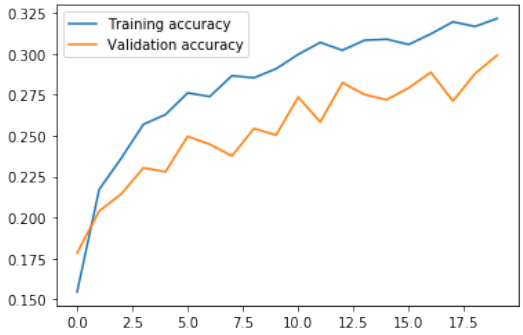


Precisión de BaseNet con `data_augmentation`

Son resultados coherentes pero nuestra precisión ha bajado por lo que elijo el anterior. Aunque ya he explicado que hacer flip vertical no tendría demasiado sentido salvo que el pájaro esté volando y la foto haya sido tomada con la cabeza apuntando a la parte inferior. Aún así siento la curiosidad por observar los resultados. Comprobemos:



Pérdida de BaseNet con `data_augmentation`



Precisión de BaseNet con `data_augmentation`

Efectivamente hemos bajado `accuracy` y hemos aumentado `loss`. Por lo tanto tras lo visto voy elegir el primer modo de aumentar los datos ya que sin ser muy sofisticado me proporciona resultados satisfactorios. Sus tasa de acierto era:

```
Test accuracy: 0.4404
```

2.3 Red más profunda

En este apartado vamos a agregar capas convolucionales y totalmente conectadas a nuestro modelo. Vamos a crear dos modelos al memos para poder comparar y discutir las diferentes mejoras. El primer modelo es:

```
def model_baseNet_mejora1(input_shape=(32,32,3)):
  model = Sequential()
  model.add(Conv2D(8, kernel_size=(5, 5), activation='relu', input_shape=input_shape))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))
  model.add(Conv2D(16, kernel_size=(5, 5), padding='same', activation='relu'))
  model.add(Conv2D(4, kernel_size=(3, 3), activation='relu'))
  model.add(Dropout(0.3))
  model.add(Flatten())
  model.add(Dense(75, activation='relu'))
  model.add(Dense(25, activation='softmax'))
  return model
```

En las capas convolucionales el primer parámetro hace referencia a la dimensión del espacio de salida (es decir, el número de filtros de salida en la convolución). He usado activación `relu` después de estas capas y en una he incluido el parámetro `padding='same'` en un capa profunda porque no quería perder más dimensiones ya que esto no se come los bordes al pasar la máscara por las imágenes.

He usado una capa `Dropout` para desactiva un número de neuronas de la red neuronal de forma aleatoria. En cada iteración de la red neuronal se desactivarán diferentes neuronas. La probabilidad de este hecho es el parámetro que le pasamos. Usaremos un valor cercano a 1 cuando queramos desactivar más neuronas y un valor cercano a 0 cuando queramos desactivar pocas. Normalmente se usan valores cercanos a 0,5 La mayor ventaja de esta capa es que reduce el sobreajuste.

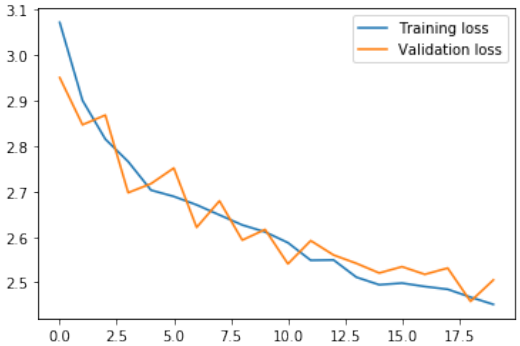
Por último he usado `Flatten` para tener un vector he añadido las capas `fully connected`, la primera con activación `relu` y posteriormente una con tantas neuronas como clases tenga el problema, 25, y una activación `softmax` para transformar las salidas de las neuronas en la probabilidad de pertenecer a cada clase.

Muestro su `summary()`:

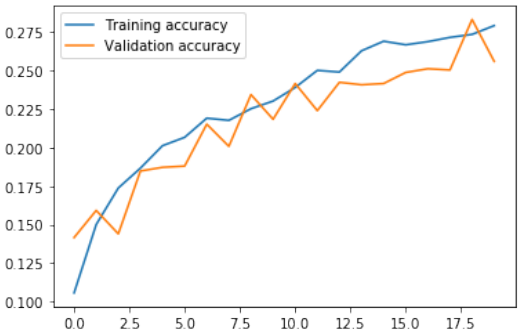
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_18 (Conv2D)           (None, 28, 28, 6)         456
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 14, 14, 6)         0
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 10, 10, 16)        2416
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 5, 5, 16)          0
_________________________________________________________________
flatten_9 (Flatten)          (None, 400)               0
_________________________________________________________________
dense_17 (Dense)             (None, 50)                20050
_________________________________________________________________
dense_18 (Dense)             (None, 25)                1275
=================================================================
Total params: 24,197
Trainable params: 24,197
Non-trainable params: 0
```

Obtenemos los siguientes resultados:

```
Test accuracy: 0.336
```



Pérdida de BaseNet red profunda



Precisión de BaseNet red profunda

No hemos colocado capas de `maxpool` después de cada capa convolucional en el modelo para no sufrir una pérdida excesiva de información. Veamos la segunda opción planteada:

```
def model_baseNet_mejora2(input_shape=(32,32,3)):
  model = Sequential()
  model.add(Conv2D(16, kernel_size=(5,5), activation='relu', input_shape=input_shape))
  model.add(MaxPooling2D(pool_size=(4,4)))
  model.add(Dropout(0.3))
  model.add(Conv2D(8, kernel_size=(5,5), activation='relu'))
  model.add(Conv2D(4, kernel_size=(3,3), activation='relu', padding = 'same'))
  model.add(Flatten())
  model.add(Dense(25, activation='relu'))
  model.add(Dense(25, activation='softmax'))
  model.summary()
  return model
```

El modelo queda así:

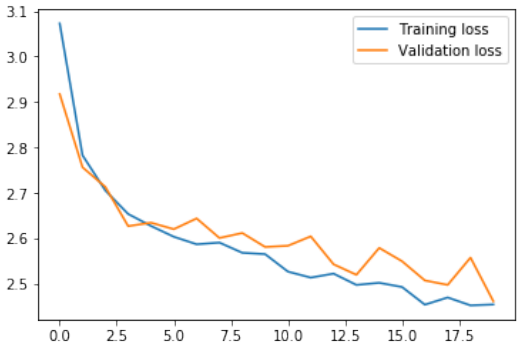
```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_32 (Conv2D)           (None, 28, 28, 16)        1216
_________________________________________________________________
max_pooling2d_23 (MaxPooling (None, 7, 7, 16)          0
_________________________________________________________________
dropout_9 (Dropout)          (None, 7, 7, 16)          0
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 3, 3, 8)           3208
_________________________________________________________________
conv2d_34 (Conv2D)           (None, 3, 3, 4)           292
_________________________________________________________________
flatten_14 (Flatten)         (None, 36)                0
_________________________________________________________________
dense_26 (Dense)             (None, 25)                925
_________________________________________________________________
dense_27 (Dense)             (None, 25)                650
=================================================================
Total params: 6,291
Trainable params: 6,291
Non-trainable params: 0
```

Obtenemos el siguiente *loss*, *accuracy* y gráficas:

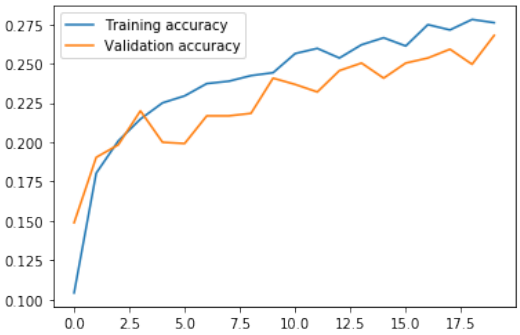
```
Test accuracy: 0.3304
```

2.4 Capas de normalización

En este apartado vamos a añadir capas de normalización a nuestro modelo. Esta capa contribuye a reducir el sobreajuste y mejorar el entrenamiento del modelo. En Keras encontramos las capas `BatchNormalization()` que se pueden incorporar fácilmente a nuestro modelo.



Pérdida de BaseNet red profunda



Precisión de BaseNet red profunda

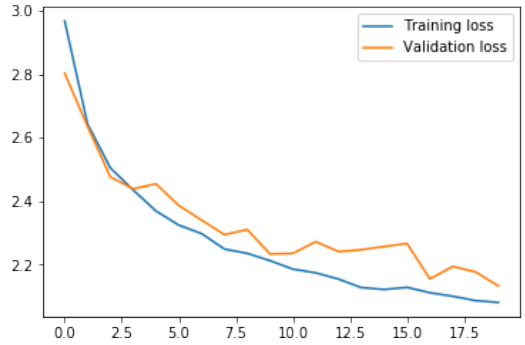
Se nos pide experimentar insertándolas antes y después de las capas `relu`. En lo que yo entiendo lo mejor es normalizar antes de hacer una operación grande como una convolución por lo que este motivo me parece bueno para poner la capa de normalización antes. Por ejemplo daría igual normalizar antes o después de `Flatten`.

```
def model_baseNet_batch(input_shape=(32,32,3)):
  model = Sequential()
  model.add(Conv2D(6, kernel_size=(5, 5), padding='same', activation='relu', input_shape=input_shape)
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.25))
  model.add(BatchNormalization())
  model.add(Conv2D(16, kernel_size=(3, 3), padding='same', activation='relu'))
  model.add(Conv2D(6, kernel_size=(3, 3), activation='relu'))
  model.add(BatchNormalization())
  model.add(Dropout(0.25))
  model.add(Flatten())
  model.add(Dense(50, activation='relu'))
  model.add(Dense(25, activation='softmax'))
  return model
```

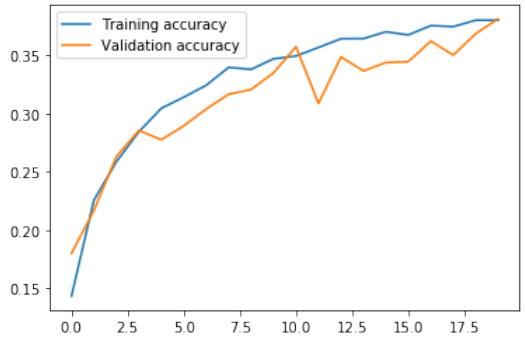
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_44 (Conv2D)           (None, 32, 32, 6)         456
_________________________________________________________________
max_pooling2d_27 (MaxPooling (None, 16, 16, 6)         0
_________________________________________________________________
dropout_15 (Dropout)         (None, 16, 16, 6)         0
_________________________________________________________________
batch_normalization_5 (Batch (None, 16, 16, 6)         24
_________________________________________________________________
conv2d_45 (Conv2D)           (None, 16, 16, 16)        880
_________________________________________________________________
conv2d_46 (Conv2D)           (None, 14, 14, 6)         870
_________________________________________________________________
batch_normalization_6 (Batch (None, 14, 14, 6)         24
_________________________________________________________________
dropout_16 (Dropout)         (None, 14, 14, 6)         0
_________________________________________________________________
flatten_18 (Flatten)         (None, 1176)              0
_________________________________________________________________
dense_34 (Dense)             (None, 50)                58850
_________________________________________________________________
dense_35 (Dense)             (None, 25)                1275
=================================================================
Total params: 62,379
Trainable params: 62,355
Non-trainable params: 24
```

Después de mostrar el `summary` vemos los resultados:

```
Test accuracy: 0.4552
```



Pérdida de BaseNet con BatchNormalization



Precisión de BaseNet con BatchNormalization

Sin duda la capa de normalización mejora los datos de una manera notables, en torno a un 10 %. También destacar que apenas encontramos sobreajuste viendo las gráficas lo cual es bueno.

2.5 *Early Stopping*

Este apartado está dedicado a responder a la : *¿Después de cuántas épocas parar y dejar de entrenar?*. Básicamente si a partir de una época empezamos a empeorar durante un número de épocas consecutivo podría interesarnos para el modelo pues estamos aumentando nuestra pérdida.

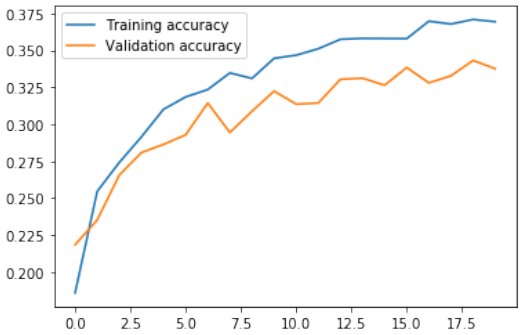
El número de épocas que estamos dispuestos a aguantar empeorando es el parámetro `patience`.

```
es = EarlyStopping(monitor='val_loss', patience=5)
hist = model.fit_generator(train_data,
          steps_per_epoch = len(x_train)*0.9/batch_size,
          epochs = epochs,
          verbose = 1,
          validation_data = validation_data,
          validation_steps = len (x_train)*0.1/batch_size,
          callbacks=[es])
```

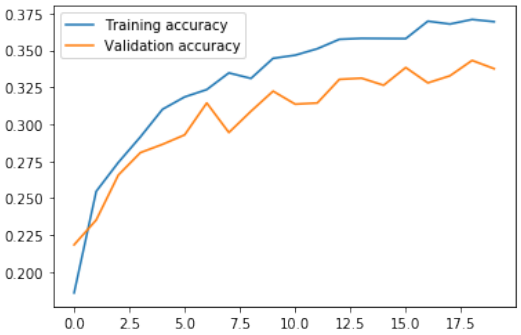
Después de mirar las gráficas de pérdida de entrenamiento y de precisión en validación he decidido escoger como número de épocas 5. Un número más pequeño que este no dejaría a mi modelo aprender lo suficiente y aumentarlo contribuiría al sobreajuste así que en el equilibrio está la virtud.

Los resultados obtenidos en este apartado son:

```
Test accuracy: 0.4456
```



Pérdida de BaseNet con `EarlyStopping`



Precisión de BaseNet con `Early Stopping`

Cuando encontramos la tasa de acierto del conjunto de validación por encimam de la del conjunto de entrenamiento significa que estamos haciendo las cosas bien. No hay mucho sobreajuste pero sí es cierto que que muestras más picos y más cambios la validación con respecto al apartado anterior.

Apartado 3

Transferencia de modelos y ajuste fino con ResNet50 para la base de datos Caltech-UCSD

El conjunto de datos a estudiar se compone de 6033 imágenes de 200 especies de pájaros. Tiene, por tanto, 200 clases, con 3000 imágenes en el conjunto de entrenamiento y 3033 en el de prueba. De nuevo, se dejará un 10 % del conjunto de entrenamiento para validación.

3.1 Resnet50

Vamos a usar la red ResNet50 como un extractor de características, usándolo preentrenado en ImageNet en el conjunto de datos mencionado anteriormente, `Caltech-UCSD`.

Lo primero es hacer el montaje. He subido un zip llamado *imágenes* a mi drive y luego aprovechba el código del esquema proporcionado. La primera vez que ejecutamos nos pide que le pasemos un código para tener permisos para entrar en nuestro drive y también:

```
Drive already mounted at /content; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

Por tanto he incluido `force_remount=True`. Del mismo modo aparecen muchos warnings por la versión de `TensorFlow`:

```
import tensorflow.compat.v1 as tf
tf.logging.set_verbosity(tf.logging.ERROR)
```

En primer lugar necesitamos haberle quitado la última capa de 1000 neuronas que clasificaba las imágenes de entrada en las clases de ImageNet. Con esto ahora tenemos un capa 2048 neuronas por lo que por cada imagen que pase por la red obtendremos un vector de tamaño 2048. Estamos pasando 3000 imágenes por lo que tenemos $3000 \cdot 2048 = 6144000$ imágenes.

Vamos a usar `ResNet50` como una caja negra y en la salida vamos a añadir dos capas fully conected en donde la última capa tiene 200 neuronas que clasificará el conjunto de imágenes que tenemos en las 200 clases que queremos.

Para quitarle la última capa a ResNet50 simplemente usamos el parámetro `include_top` igualándolo a `False`. Por otro lado tenemos que pasarle a los datos la función de preprocesamiento indicada a través de `ImageDataGenerator` tanto en el conjunto de entrenamiento como en el conjunto de test.

```
resnet50 = ResNet50(weights='imagenet', include_top=False, pooling="avg",
                    input_shape=(224, 224, 3))
datagen_train = ImageDataGenerator (preprocessing_function=preprocess_input)
datagen_test = ImageDataGenerator(preprocessing_function=preprocess_input)
```

La función `predict_generator` genera predicciones para las muestras de la entrada de datos del generador, tanto de los datos del conjunto de entrenamiento como del conjunto de test.

Las características extraídas en el paso anterior son la entrada del nuevo modelo con las dos capas `fully connected` en el que la útlima tiene 200 neuronas por las 200 clases que tienen los datos `Caltech-UCSD`:

```
def modeloAp1():
  model = Sequential()
  model.add(Dense(1024, activation='relu', input_shape=(2048, )))
  model.add(Dense(200, activation='softmax'))
  return model
```

Ya que este modelo tendrá como entrada la capa de 2048 neuronas que proporciona el `ResNet50`, ponemos como `input_shape` 2048. Hemos usado `SGD`, Gradiente Descendente Estocástico, en nuestro compilador. Para entrenar podemos usar `fit` (indicándole `validation_split=0.1`) y `fit_generator=0.1`

```
def extraerCaracteristicas():
  input_shape=(224, 224, 3)    # Imágenes en color con 3 canales de 224x224 píxeles.
  batch_size = 32              # Tamaño de batch potencia de 2
  epochs = 15                  # Elegimos número de épocas

  print("Leyendo datos\n")
  x_train, y_train, x_test, y_test = cargarDatos("/content/images")

  # Creamos un generador para entrenamiento y otro para test
  datagen_train = ImageDataGenerator(preprocessing_function = preprocess_input)
  datagen_test = ImageDataGenerator(preprocessing_function = preprocess_input)

  # Usamos ResNet50 preentrenada en ImageNet sin la última capa
  resnet50 = ResNet50(weights='imagenet', include_top = False, pooling = "avg", input_shape = (224, 2

  # Extraemos características de las imágenes con el modelo anterior
  print("Extraemos ícaractersticas\n")
  features_train = resnet50.predict_generator(datagen_train.flow(x_train, batch_size = 1, shuffle = 
  features_test = resnet50.predict_generator(datagen_test.flow(x_test, batch_size = 1, shuffle = Fals

  # Compilamos el modelo
  model = modeloAp1()
  opt = SGD(lr = lr, decay = 1e-6, momentum = 0.9, nesterov = True)
  model.compile(loss = categorical_crossentropy, optimizer = opt, metrics = [acc])

  features_train = ImageDataGenerator(validation_split=0.1)
  weights = model.get_weights()
  model.set_weights(weights)
  model.summary()

  # Entrenamos el modelo
  hist = model.fit(features_train, y_train,
                   batch_size = batch_size,
                   epochs = epochs,
                   verbose = 1,
                   validation_split = 0.1)
```

```
mostrarEvolucion(hist)
score = model.evaluate_generator(datagen_test.flow(x_test, y_test, batch_size = 1, shuffle = False)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

Tengo un problema en *Google Colab* y me tarda muchísimo en ejecutar la función anterior por lo que no vamos a poder observar los resultados, esperabamos una tasa de acierto cercana del 50 %

3.2 *Fine-tuning* ResNet50

Ahora vamos a usar un modelo residual, residual es que no es un modelo secuencial. Ahora no voy de una capa a solo una y de ahí solo a otra. No, ResNet se mueve entre capas. En `keras` está implementado por la clase `Model`.

En este apartado ya no tenemos la red preentrenada, lo vamos a hacer nosotros para que clasifique nuestro problema. Parte de la estructura del problema y ciertas explicaciones de qué hace cada función es exactamente como en el apartado anterior.

Aquí nuestro modelo es:

```
def modeloAp2(x):
  x = Dense(512, activation='relu')(x)
  x = Dropout(0.6)(x)
  last = Dense(200, activation='softmax')(x)
  return last
```

He añadido una capa `Dropout` para desactivar el 60 % de neuronas (quitamos sobreajuste).

La función que ejecuta todo lo pedido en este apartado es:

```
def ejercicio3_2(batch_size, epochdef ejercicio3_2(batch_size, epochs):
  input_shape=(224, 224, 3)    # Imágenes en color con 3 canales de 224x224 píxeles.
  batch_size = 32              # Tamaño de batch potencia de 2
  epochs = 15                  # Elegimos número de épocas

  print("Leyendo datos\n")
  x_train, y_train, x_test, y_test = cargarDatos("/content/imagenes")

  # Creamos un generador para entrenamiento y otro para test
  datagen_train = ImageDataGenerator(preprocessing_function = preprocess_input)
  datagen_test = ImageDataGenerator(preprocessing_function = preprocess_input)

  # Usamos ResNet50 preentrenada en ImageNet sin la última capa
  resnet50 = ResNet50(weights='imagenet', include_top=False, pooling="avg", input_shape=(224, 224, 3)

  # Reentrenamos el modelo
  last = modeloAp2(resnet50.output)
  new_model = Model(inputs=resnet50.input, outputs=last)

  # Compilamos el modelo
  print("Compilando el modelo\n")
  opt = SGD(lr=lr, decay=1e-6, momentum=0.9, nesterov=True)
  new_model.compile(loss = categorical_crossentropy, optimizer=opt, metrics=[acc])
```

```
features_train = ImageDataGenerator(validation_split=0.1)
weights = new_model.get_weights()
new_model.set_weights(weights)
new_model.summary()

# Entrenamos el modelo
hist = model.fit(features_train, y_train,
                 batch_size=batch_size,
                 epochs=epochs,
                 verbose=1,
                 validation_split=0.1)

mostrarEvolucion(hist)
score = new_model.evaluate_generator(datagen_test.flow(x_test, y_test, batch_size = 1, shuffle = F
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

Hemos compilado y entrenado igual que antes y por último evaluamos nuestro modelo con el test.