# Lez15_YOLO

November 24, 2023

## 1 Lezione 15 - YOLO

Ci siamo lasciati con *object detection*. Siamo passati a reti *completamente convoluzionali*. Se fatto bene, la rete ottenuta è equivalente. Il vantaggio principale è che tale rete funziona anche con immagini più grandi di quelle fornite durante l'addrestramento.

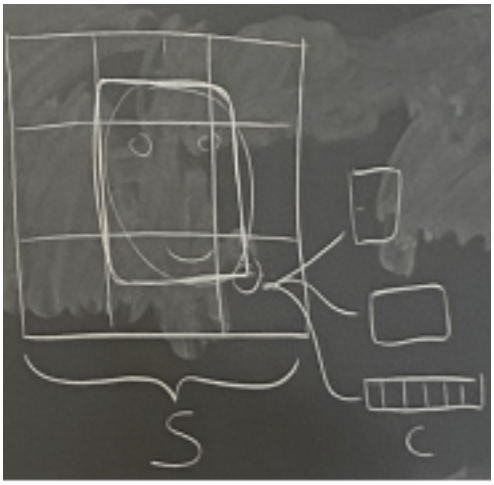
### 1.1 YOLO - You Only Look Once

Vediamo l'oggetto una sola volta, per fare object detection. Inizialmente era molto veloce, ma non troppo accurato. Esistono svariate versioni di *Yolo* disponibili e già pronte.

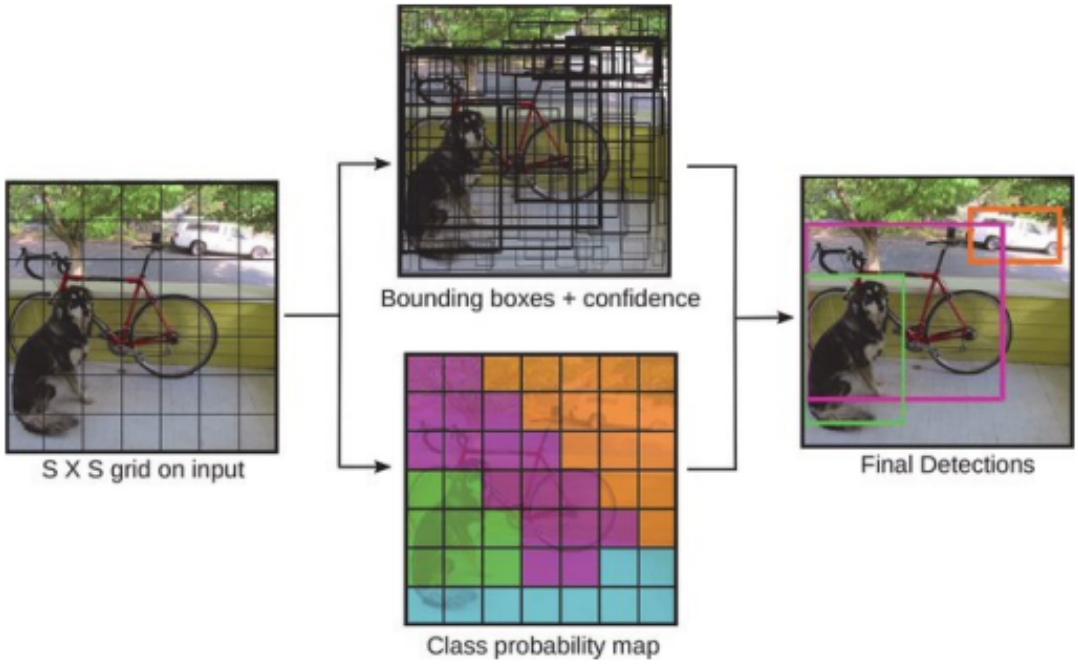
Il fatto di vedere l'immagine una sola volta fornisce il beneficio di vedere l'immagine nella sua interezza. Un oggetto grande è visto nella sua totalità, senza *spezzarlo*. C'è comunque una griglia *logica*. Più celle ho, più bounding box predico. Non c'è però un confine rigido, quindi, se ci troviamo in una cella $(x, y)$, possiamo far riferimento alle celle adiacenti per la valutazione. La rete non ha *paletti* su questo. Per ciascuna cella, ci sono $B$ predizioni *bounding box* (2 in $\text{YOLO}_1$) e una probabilità di appartenenza alle $C$ classi della detection. Per ogni cella avremo una classe dominante. Ogni *bounding box* presenta 5 valori associati:

- coordinate $(x, y)$
- altezza e larghezza $h, w$
- confidence score

Il bounding box non è detto che sia *dentro* la cella. Ad esempio, se un volto umano è racchiuso in più celle, è possibile che una cella che ne vede solo una parte, possa comunque generare un bounding box. Il *confidence score* ci dice della fiducia del bounding box nell'aver circoscritto un oggetto al suo interno.



In questa immagine, ad ogni colore corrisponde una certa classe dominante, per la *class probability map*. Vengono generati anche bounding box (immagine sopra), probabilmente molti identificano lo stesso oggetto.




Ciò che faccio è associare tutti i bounding box alla probabilità della classe, moltiplicata per la confidence.

Come tengo i migliori?

Sfruttiamo *interception over union*.

Dopo aver generato $S \times S \times B$ bounding box, l'IoU (Intersection over Union) viene utilizzato per eliminare rilevamenti ridondanti dello stesso oggetto.

- Se due scatole hanno un alto IoU, è probabile che abbiano rilevato lo stesso oggetto.
- Si sceglie quella con la maggiore confidenza.
- *Durante l'addestramento*, l'IoU viene utilizzato anche per calcolare i punteggi di confidenza: $P(\text{Oggetto}) \times \text{IoU}_{\text{pred}}^{\text{truth}}$ dove $P(\text{Oggetto}) \in \{0, 1\}$, a seconda che un oggetto centrato nella cella esista o meno.

Scartiamo anche i bounding box inferiori ad un certo *lower bound*. Come vediamo, le probabilità sono slegate dalla ricerca dei bounding box, questo perchè "mettere tutto insieme" (ad esempio: individuo l'oggetto, vedo il bounding box e poi dico che tipo di oggetto è) risulterebbe computazionalmente più complesso.

Perchè produciamo $n$ bounding box? Alla fine solo uno sarà quello corretto. Avendo tante celle e tanti bounding box prodotti, una cella adiacente può usare le nostre valutazioni per migliorare la sua valutazione.

$\text{YOLO}_1$ usava 24 livelli di convoluzione e 2 livelli completamenti connessi. Esistono anche varianti più leggere. Solo da $\text{YOLO}_2$ si è passati a livelli di convoluzione completamente connessi.

Abbiamo detto che c'è un calcolo disaccoppiato tra classi e bounding box. In $\text{YOLO}_3$, vengono suggeriti dei rapporti nelle forme degli oggetti. A ciò viene applicato l'algoritmo di cloustering $K - Means$, in cui troviamo delle forme tipiche (ad esempio: pedoni nelle immagini della strada). Questi valori saranno relativi ad *anchor boxes*. *Suggeriamo* alla rete di trovare oggetti con tale forma.

Sono a disposizioe le versioni 3 e 4 di YOLO a:

https://pjreddie.com/darknet/yolo/

Fatto in `C`, in quanto si cercava la velocità. La versione 5 è reperibile su:

https://github.com/ultralytics/yolov5

sfrutta `PyTorch`, ma non è un problema in quanto noi dobbiamo usarlo, non modificarlo.

Il tutto è basato su `ImageNet`, che contiene numerosi oggetti.

YOLO è anche usato (insieme ad altre cose) nelle partite di *calcio*, per riconoscere palloni, calciatori, arbitri. Come? prendendo video di partite di calcio e fare *Transfer Learning*. Può essere usato anche in *diretta*. (Esempi con $\text{YOLO}_5$). La colonna `mAP` ci dice l'accuracy, meglio puntare ad una accuracy non troppo elevata se non abbiamo *GPU*.

## 1.2 Object Tracking

Lavorare su un video vuol dire distinguere tra frame diversi se abbiamo una stessa persona o due persone diverse. Oltretutto per ogni frame compie del lavoro. Magari a noi basta, individuata una persona, percorrere i suoi movimenti. Per far ciò esistono *framework* appositi, come `DeepSORT`. Esso usa i filtri di *Karman* per predire i suoi spostamenti, in modo da sapere dove mi aspetto di vedere l'oggetto nel prossimo frame.

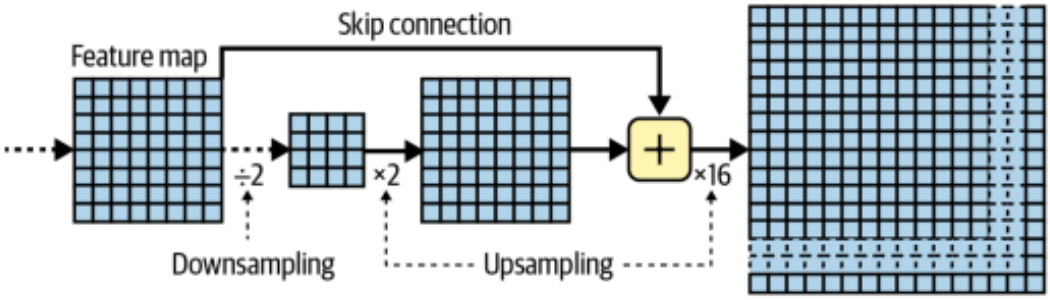
Una implementazione di YOLo ft DeepSORT è reperibile qui:

https://github.com/theAIGuysCode/yolov4-deepsort

### 1.3 Semantic Segmentation

Trattasi di *detection* a grana fine. Non ci basta il box, vogliamo esattamente i bordi. Non è in grado di identificare in modo singolo, due oggetti dello stesso tipo vicino. Nell'immagine, infatti, con `cars` identifichiamo più macchine insieme, in quanto aventi bordi che si toccano.


Una prima parte della rete viene usata per fare *Detection*. Arrivati ad un certo punto della rete, essa si è fatta un'idea su dove si aspetta questi elementi. Successivamente mappa questi oggetti sulla foto originale, mediante una tecnica di *upscaling* (esempio: moltiplico per 16) mediante un livello convoluzionale inverso, che parte da un qualcosa di piccolo e la ingrandisce. Come si fa? Impostando un *fractional stride* $\leq 0.5$, ovvero spostandoci di un numero pari a meno di un pixel (es: mi sposto di mezzo pixel). Da sola questa cosa non basta, si usano anche *skip connections*, nella quale tengo conto dei pixel nella risoluzione originale. E' comunque un qualcosa slegata da ciò che vediamo noi, quindi skippabile.



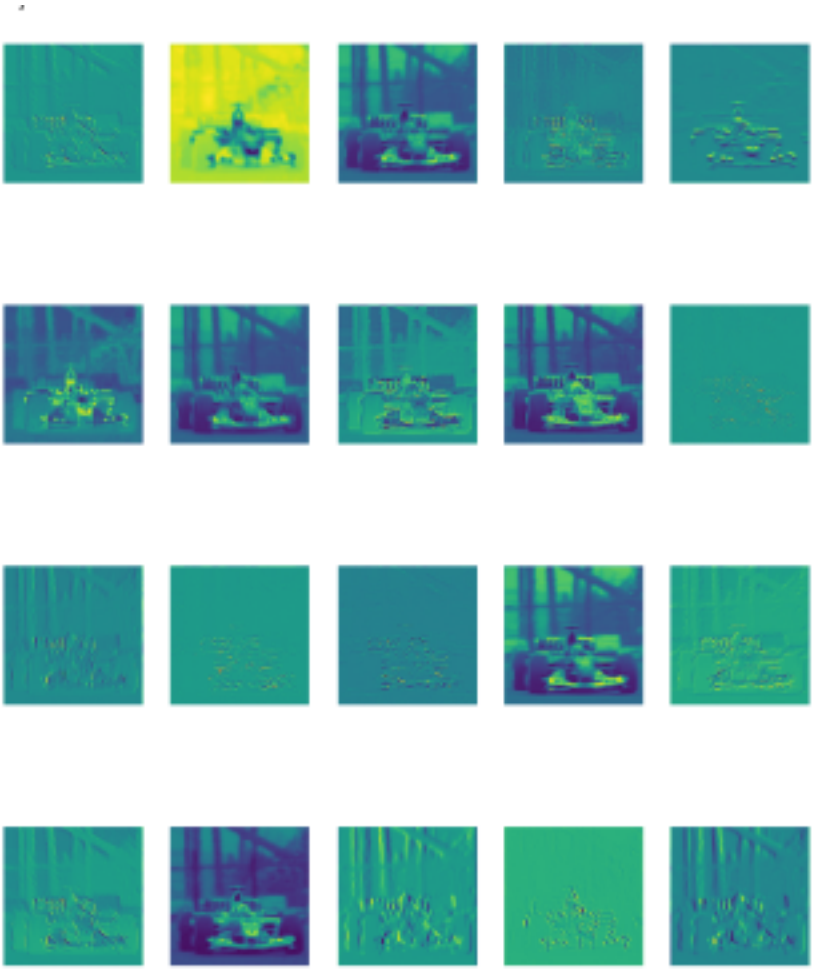

## 2 Parte Coding

### 2.1 Come fa la rete a riconoscere gli oggetti?

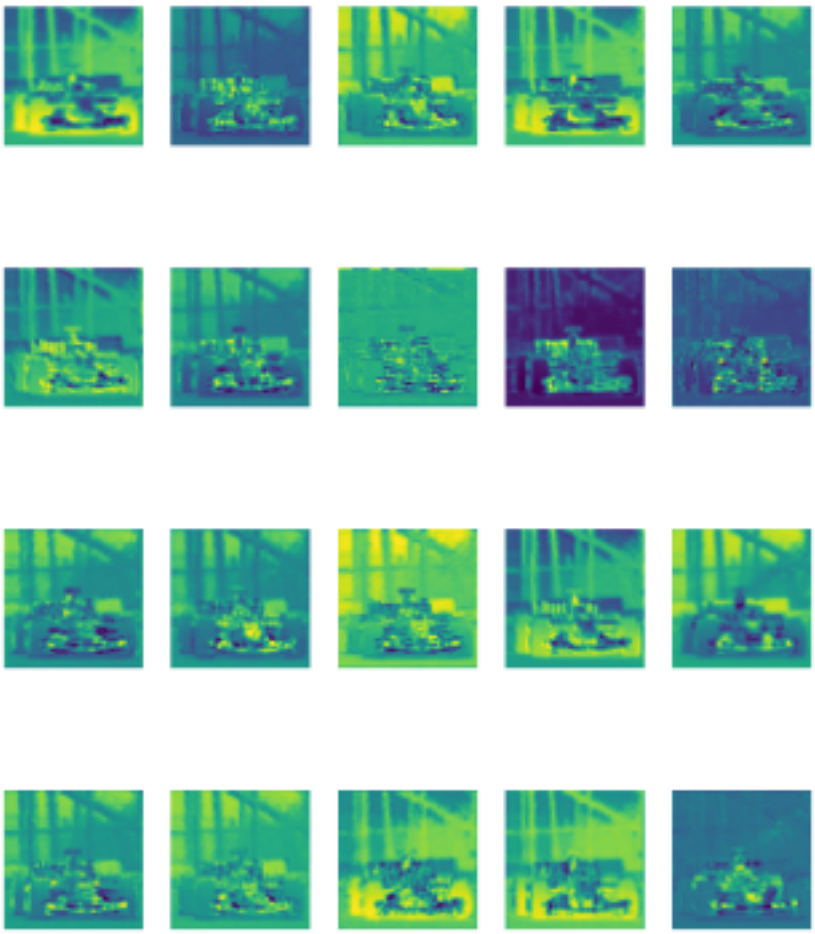
La risposta è nelle *features*. I filtri, cosa permettono di vedere alla rete? Vediamolo:

### 2.1.1 Layer 1

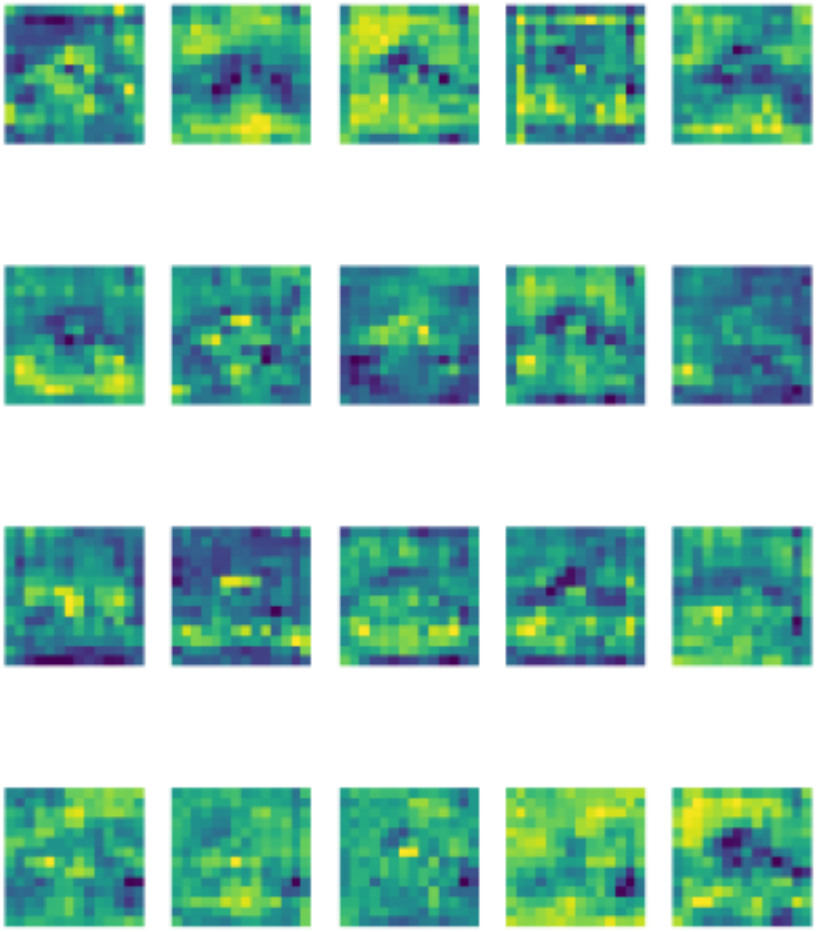
### 2.1.2 Layer 3



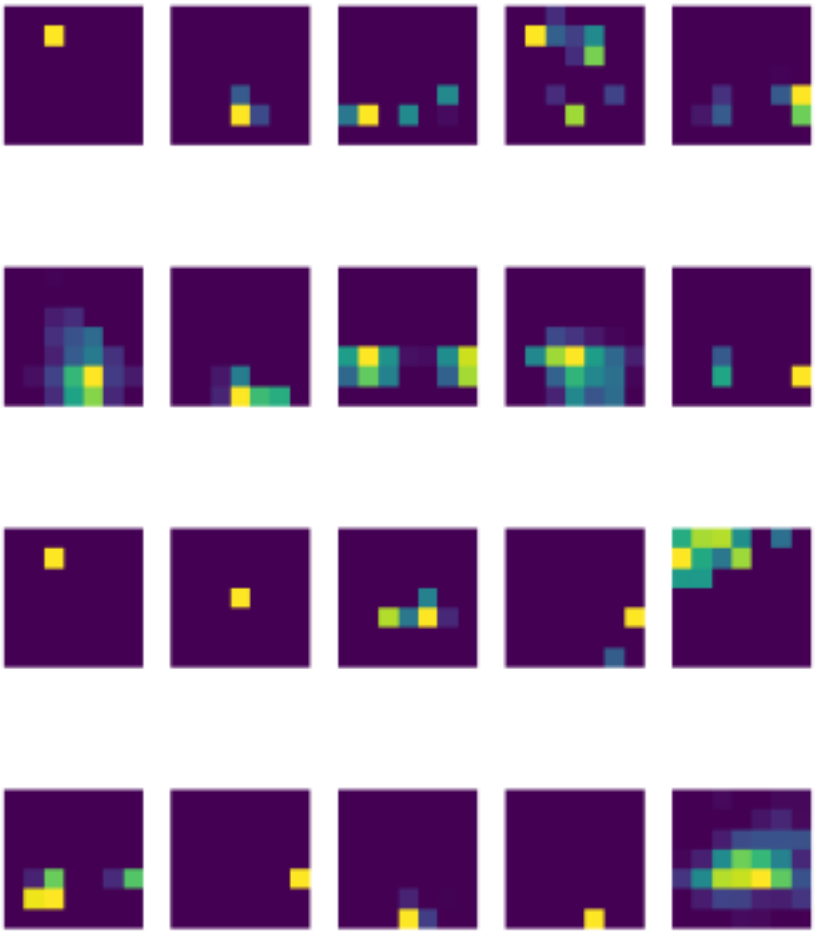
### 2.1.3 Layer 15



#### 2.1.4 Layer 130



### 2.1.5 Layer 174



In ogni features map abbiamo informazioni che sembrano poco rilevanti, ma non è cosi!

## 2.2 Classificatore cavallo binario - transfer_horses.ipynb

L'obiettivo è capire dove si trova il cavallo. Usiamo ResNet50, addrestrata su *ImageNet*. Prendiamo rete che già esiste. La classificazione è {positiva, negativa}, cioè c'è o non c'è un cavallo.

```
1/1 [==============================] - 0s 458ms/step
Image #0
  n03388043 - fountain     43.82%

1/1 [==============================] - 0s 54ms/step
Image #0
  n01582220 - magpie       18.02%

1/1 [==============================] - 0s 54ms/step
Image #0
  n02422106 - hartebeest   35.62%

1/1 [==============================] - 0s 57ms/step
Image #0
  n02109047 - Great_Dane   17.06%
```

In ImageNet non ci sono cavalli, quindi non vengono classificati. Dobbiamo *aggiungerlo noi*. Mediante

```
train_ds = tf.keras.utils.image_dataset_from_directory(
  DATASET_DIR,
  validation_split=0.2,
      subset="training",
  seed=123,
  image_size=(224, 224),
  batch_size=32)
validation_ds = tf.keras.utils.image_dataset_from_directory(
    DATASET_DIR,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(224, 224),
  batch_size=32)
```

Associamo a delle cartelle con delle immagini le immagini di ciò che vogliamo classificare.


### 2.3 Svolgimento

```
base_model = keras.applications.ResNet50(
    weights="imagenet",  # Load weights pre-trained on ImageNet.
    input_shape=(224, 224, 3),
    include_top=False, #non voglio l'ultimo livello
)  # Do not include the ImageNet classifier at the top.

# Create new model on top
inputs = keras.Input(shape=(224, 224, 3))

# Pre-trained Xception weights requires that input be scaled
# from (0, 255) to a range of (-1., +1.), the rescaling layer
# outputs: `(inputs * scale) + offset`
scale_layer = keras.layers.Rescaling(scale=1 / 127.5, offset=-1)
x = scale_layer(inputs)
```

```
# The base model contains batchnorm layers. We want to keep them in inference mode
# when we unfreeze the base model for fine-tuning, so we make sure that the
# base_model is running in inference mode here.
x = base_model(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dropout(0.2)(x)  # Regularize dropout, passiamo al complet. connesso
outputs = keras.layers.Dense(1)(x) #devo solo dire se è o meno cavallo.
model = keras.Model(inputs, outputs)

#non è specificato liv. attivazione, poichè incluso in Loss.

model.summary()
#no funzione sigmoid alla fine, risultato va da -inf a +inf, ci pensa poi la loss.

model.compile(
    optimizer=keras.optimizers.Adam(),
    loss=keras.losses.BinaryCrossentropy(from_logits=True), #qui è incluso liv. attivazione
    metrics=[keras.metrics.BinaryAccuracy()],
)

epochs = 15
model.fit(train_ds,
          epochs=epochs,
          validation_data=validation_ds,
          callbacks=[keras.callbacks.EarlyStopping(patience=3)])
```

Otteniamo `accuracy = 0.7647` e `loss = 0.5338`

### 2.4 Fine Tuning

Fino ad ora abbiamo addestrato solo gli ultimi livelli, non i primi. Cerchiamo di sbloccare i pesi dei vari livelli, non ci limitiamo all'ultimo livello. Modifichiamo questo parametro (prima non era presente, o comunque non esplicitato a `True`.

```
# Freeze the base_model
base_model.trainable = True
```

e successivamente modificando il `learning_rate` ad un valore basso.

```
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate = 0.00005), #impongo lui
    loss=keras.losses.BinaryCrossentropy(from_logits=True),
    metrics=[keras.metrics.BinaryAccuracy()],
)
```

Con 10 epoche, abbiamo `accuracy = 0.94` e `loss = 0.0344`

### 2.5 Addestramento da 0

Se lo addrestrassi da 0? Non c'è più il parametro `weights`, quindi non è pre-addestrato.

```
# Create new model on top
inputs = keras.Input(shape=(224, 224, 3))

# Pre-trained Xception weights requires that input be scaled
# from (0, 255) to a range of (-1., +1.), the rescaling layer
# outputs: `(inputs * scale) + offset`
scale_layer = keras.layers.Rescaling(scale=1 / 127.5, offset=-1)
x = scale_layer(inputs)

# The base model contains batchnorm layers. We want to keep them in inference mode
# when we unfreeze the base model for fine-tuning, so we make sure that the
# base_model is running in inference mode here.
x = keras.applications.ResNet50(
    input_shape=(224, 224, 3),
    include_top=False,
) (x)
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dropout(0.2)(x)  # Regularize with dropout
outputs = keras.layers.Dense(1)(x)
newmodel = keras.Model(inputs, outputs)

newmodel.summary()
```

Il problema è che stiamo addestrando 23 milioni di parametri con 272 campioni, producendo un *overfitting* importante. Il modello impara subito a capire, nelle 272 immagini, quali presentano un cavallo e quali no. Non ha capito però le features per riconoscerne altre. Magari, vedendo le foto, ha capito che un certo pixel $(x, y)$ è sempre associato alla figura di un cavallo, ma ovviamente ciò non basta! E' una rete così potente che per queste poche immagini, neanche perde tempo ad apprendere le loro features, ma impara a memoria quali immagini presentano cavalli e quali no. Questo ci fa capire l'importanza del *Transfer Learning*.