Lez14_LeNetDerivazioni2

November 22, 2023

1 Lez_14 LeNetDerivazioni2

1.1 Recap

Nella precedente lezione, abbiamo visto la possibilità di *riusare* modelli già pronti, andando ad *sostituire* il livello di uscita precedente(dipendente dalle classi usate nel modello originale) con un livello di uscita più attinente a ciò che stiamo facendo *ora*. Tramite *Fine-Tuning* possiamo *scongelare* l'intero modello e riaddestrarlo con i nuovi dati, mediante un *learning rate basso*.

Nell'esempio di **Xception**, vogliamo sfruttare un classificatore avente 1000 classi, riadattandolo per lavorare con 2 classi.

1.2 Esempio applicativo

```
layer = keras.layers.Dense(3)
layer.build((None, 4))  # Create the weights
layer.trainable = False  # Freeze the layer, non subisce il `FIT`

print("weights:", len(layer.weights))
print("trainable_weights:", len(layer.trainable_weights))
print("non_trainable_weights:", len(layer.non_trainable_weights))
```

L'output sarà di due parametri non addestrabili:

```
weights: 2
trainable_weights: 0
non_trainable_weights: 2
```

Ciò può essere fatto anche sul modello direttamente.

1.2.1 BatchNormalization

Presenta differenze tra la parte *training* e quella di *testing*. Centra il valore medio delle attivazioni in 0, prendendo in input il mini batch e sottraendo il valore medio del mini batch. Nel test, non calcolo su ogni nuovo dato `media` e `varianza`, ma uso quelle calcolate nel `training`, fissati durante il training e non riaggiornati. Ciò si traduce nel fatto che, impostando `trainable=false` nel livello di BatchNormalization, il comportamento sarà come nella fase di inferenza.

La Batch Normalization contiene 2 pesi non addestrabili che vengono aggiornati durante l'addestramento. Queste sono le variabili che tengono traccia della media e della varianza degli input. Quando impostiamo `bn_layer.trainable = False`, lo strato di Batch Normalization funzionerà in modalità di inferenza e non aggiornerà le sue statistiche di media e varianza. Questo non è il caso di altri layer in generale, poiché la possibilità di addestramento dei pesi e le modalità di inferenza/addestramento sono due concetti ortogonali. Ma i due sono legati nel caso dello strato di Batch Normalization. Quando scongeliamo un modello che contiene strati di Batch Normalization per fare del *fine-tuning*, dobbiamo mantenere gli strati di Batch Normalization in modalità di inferenza passando `training=False` quando chiamiamo il modello di base. Altrimenti, gli aggiornamenti applicati ai pesi non addestrabili distruggeranno improvvisamente ciò che il modello ha imparato.

I dati con cui lavoriamo sono:

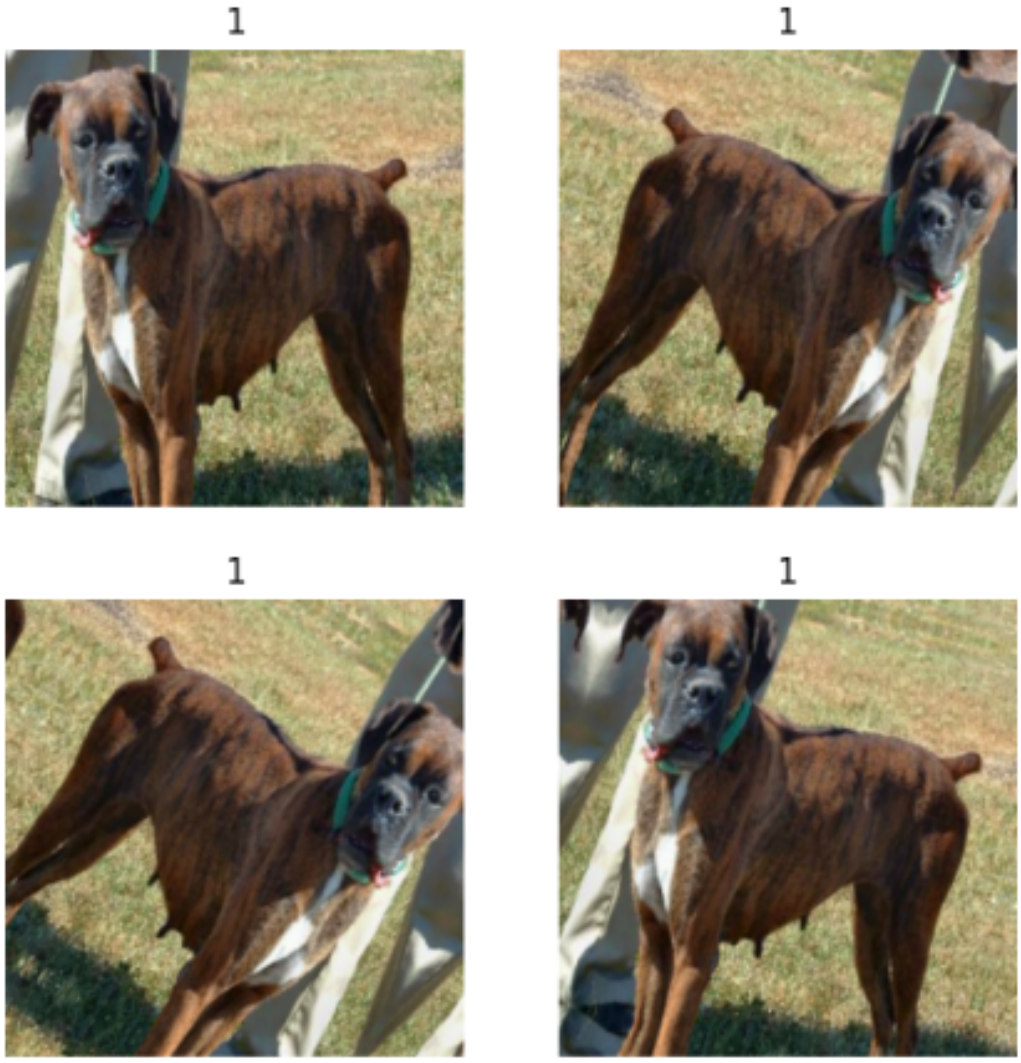
La size è diversa, dobbiamo *standardizzare*, mediante:

```
size = (299, 299)
```

```
train_ds = train_ds.map(lambda x, y: (tf.image.resize(x, size), y))
validation_ds = validation_ds.map(lambda x, y: (tf.image.resize(x, size), y))
test_ds = test_ds.map(lambda x, y: (tf.image.resize(x, size), y))
```

Per aumentare i dati, usiamo funzioni di **Keras** per il flip e la rotazione:

```
from tensorflow import keras
from keras import layers

data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
    ]
)
```

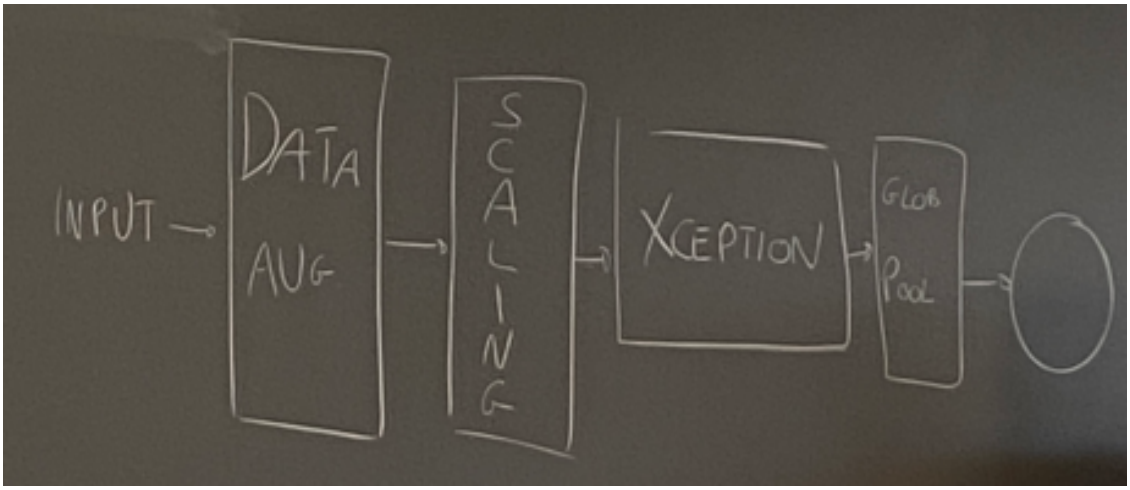


Queste sono alcune delle trasformazioni che possiamo ottenere.

1.2.2 Caricamento modello base

```
base_model = keras.applications.Xception(
    weights="imagenet",  # Load weights pre-trained on ImageNet.
    input_shape=(299, 299, 3), #dimensione input, 3 canali per ogni img.
    include_top=False, #cancelliamo gli ultimi livelli della rete,
    #lasciando solo livelli convoluzionali
)

# pesi congelati
base_model.trainable = False

# modello completo = base + altri livelli compl. connessi
inputs = keras.Input(shape=(299, 299, 3)) #tensore input
x = data_augmentation(inputs)  # vede un dato, genera versione diversa random.

# 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(x)
x = base_model(x, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dropout(0.2)(x)  # Regularize with dropout
outputs = keras.layers.Dense(1)(x) #un neurone
model = keras.Model(inputs, outputs)

model.summary()
```

Poichè classificazione binaria, mi basta un solo neurone. Gli step sarebbero:



Manca la funzione di attivazione, attualmente il valore sarebbe tra $-\infty$ e $+\infty$

In realtà, ciò viene calcolata nella funzione di *loss*, non deve per forza essere *interna* al training.

Sicuramente, specificandola, è meglio rispetto a farlo così!

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

epochs = 5 # 10+ would be better
model.fit(train_ds, epochs=epochs, validation_data=validation_ds)
```

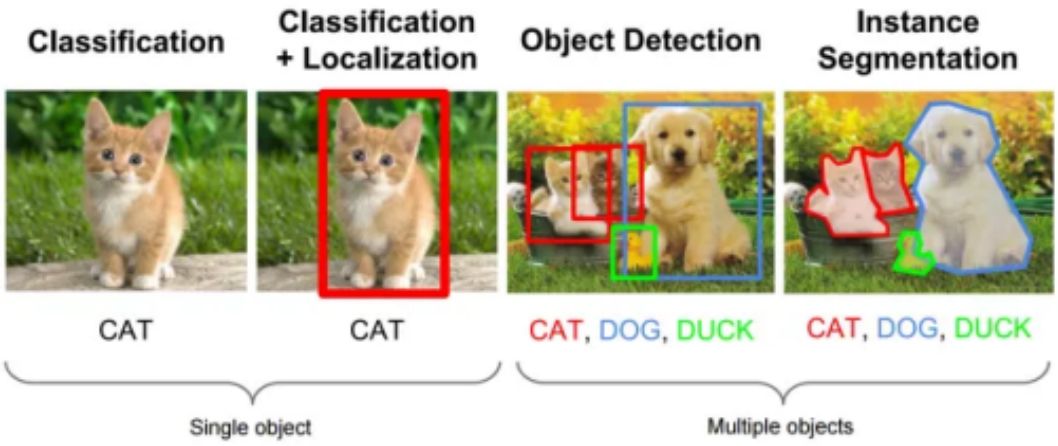
Ogni epoca richiederebbe circa 25 seconda, con prima accuracy del 95%, e quella finale 99%, che troviamo anche applicando il tutto al *test-set*.

1.2.3 Se non avessi usato Transfer Learning?

Facendo la semplice predizione,l'unica modifica è data dal fatto che il modello riconosce più razze di cani, che quindi dobbiamo *unire*. L'accuracy è del 92%, ottima, ma minore di quella appena discussa.

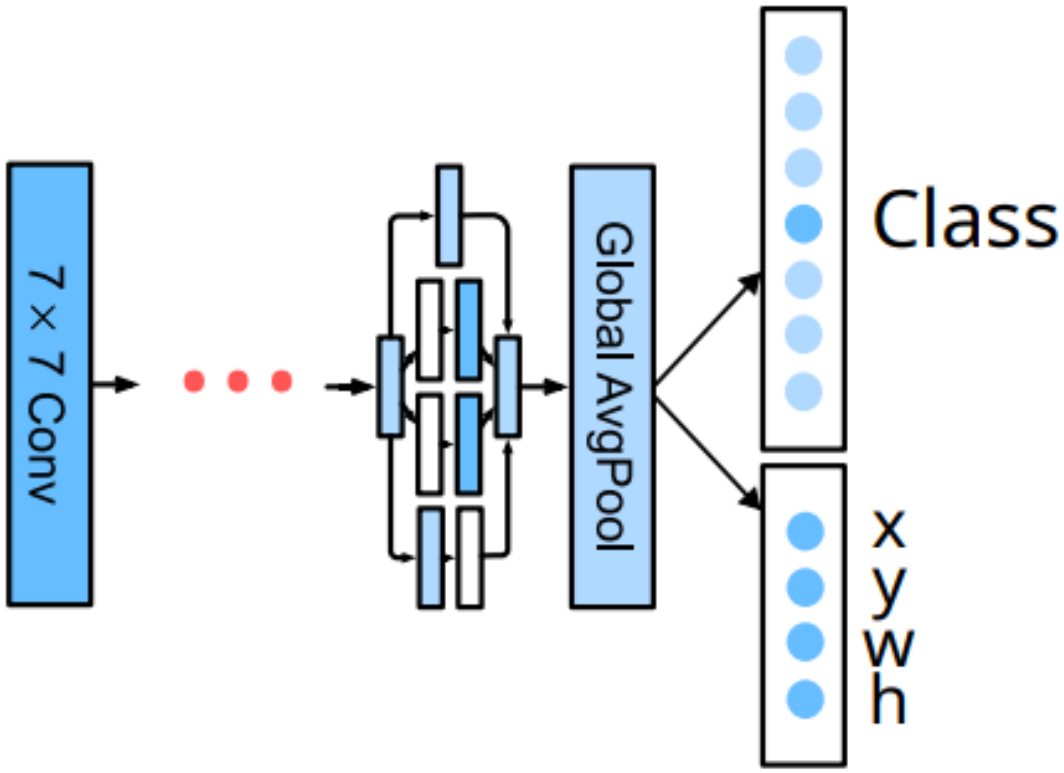
1.3 Beyond Classification

Fino ad ora, il problema era del tipo: "*associo figura ad una delle N classi*". Possiamo fare anche altro, in particolare:



1.3.1 Localization

Deve fornirci anche *bounding box*, ovvero coordinate (x, y), *altezza*, *larghezza*. Problema di regressione, devo predire quattro valori reali. Per individuarlo, prima applico la classificazione. Quindi applichiamo due livelli allo stesso tensore output. Ovvero:



Come lo scrivo in `Keras`?

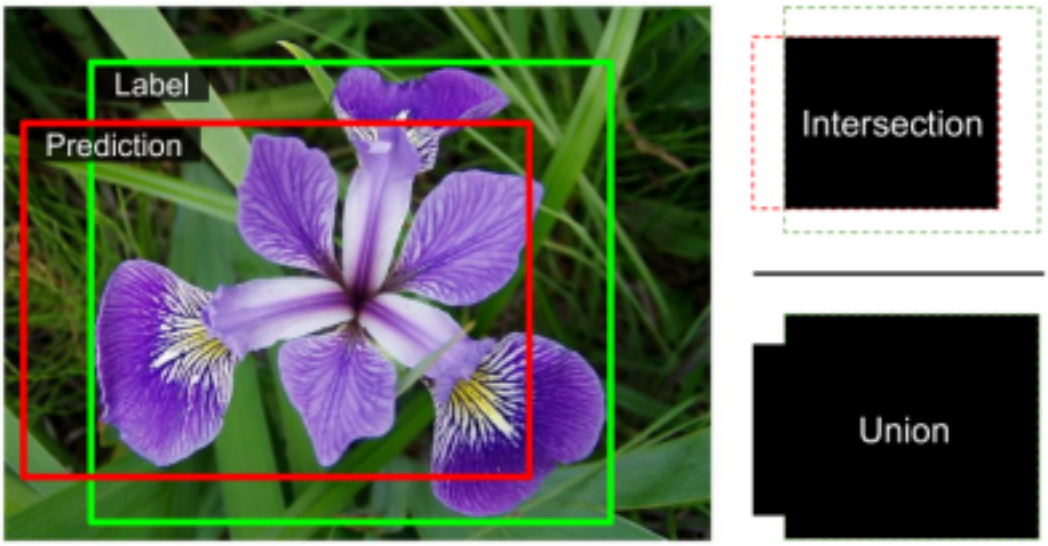
“‘python base = applications.xception.Xception (weig ts= imagene ” , p = F) avg = GlobalAveragePooling2D()(base.output) cls_out = Dense(N, activation=“softmax”)(avg) loc_out = Dense(4)(avg) model = Model(inputs=base.input, outputs=[cls_out, loc_out]) model.compile(loss=[“sparse_categorical_crossentropy”, “mse”], loss_weights=[0.8, 0.2]#nel dubbio, 50 e 50, loss tot = somma loss, optimizer=optimizer, metrics=[“accuracy”])““

Come trovo le coordinate del bounding box? E’ la parte più complessa, perchè i *bounding box* vengono presi… a mano! Esistono software che aiutano a fare ciò, però sempre la parte umana ci vuole. Esistono siti, in cui paghi per far etichettare immagini ad altri. (`Amazon Mechanical Turk`)

La predizione del bounding box è influenzata dalla classe che deve essere predetta? Ovvero, il box di un essere umano in piedi, me lo aspetto diverso rispetto a quello di una tigre. Nel modello, questo aspetto non viene considerato (per ora).

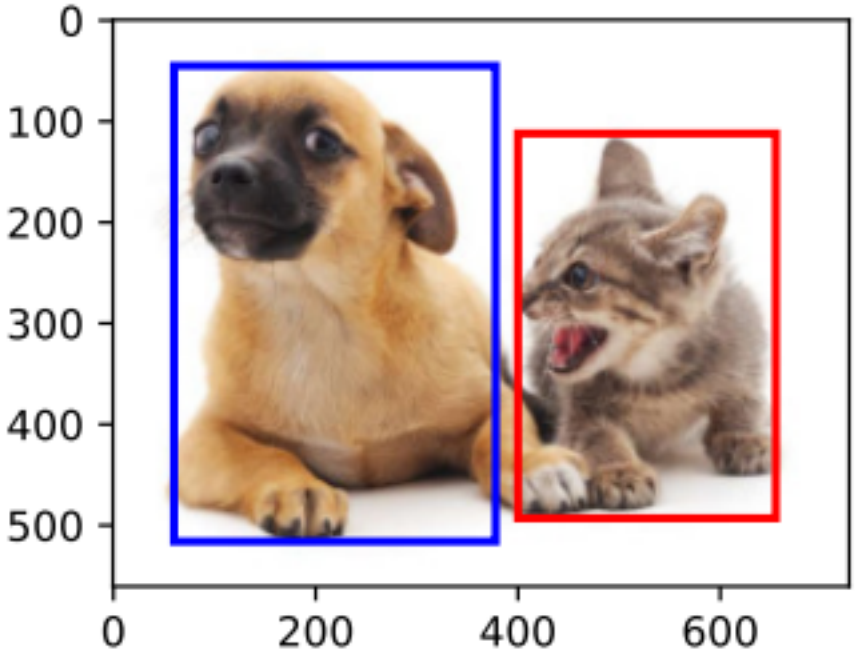
1.3.2 Metrics

Usiamo l’**errore quadratico medio** per l’addestramento. Per la *valutazione* di un bounding box, si usa **Intersection over union**. Prendo la predizione, prendo il bounding box corretto, ne calcolo intersezione ed unione, facendo unione. Se coincidono, l’intersezione sarà uguale all’unione, e il rapporto sarà 1.



1.4 Object Detection

Se avessi più di un soggetto nell'immagine? Non so a priori quanti soggetti ci siano!



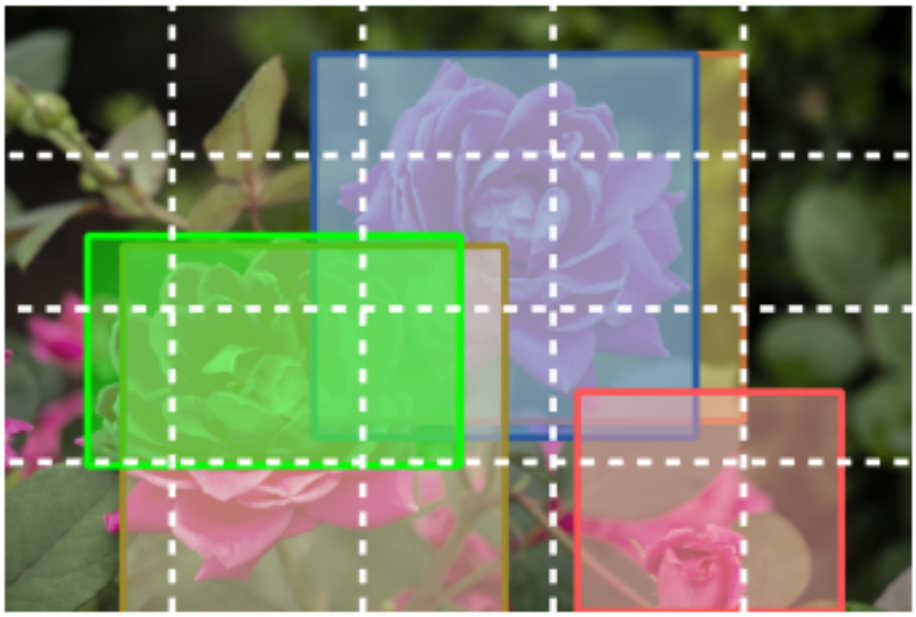
L'idea è **Naive Sliding Window Approach**, ovvero alleniamo per una singola *localizzazione*, e poi faccio scorrere la rete convoluzionale lungo l'immagine. E' presente un ulteriore output, che potrebbe appartenere al livello *dense*. Questo **Confidence score** è un numero reale che ci dice la probabilità che ci sia un certo soggetto nell'immagine, con una certa probabilità. Da dove estraggo questa *confidenza*? L'idea sarebbe prenderle dalle probabilità di appartenere ad una certa classe.

Ovvero:

- Se ho classificazione 99% di un cane, sono confidente.
- Se ho classificazione 51% di un cane, sono molto meno confidente.

1.4.1 Sliding Windows

Nella figura, identifichiamo più *Bounding Box*, sono tutti utili? Nella figura,prendendo la rosa in basso a sinistra, abbiamo due bounding box (*verde* e *rosa*), vedo il valore *intersection over union*, hanno trovato stesso oggetto, allora prendo quello la *bounding box* con confidenza più alta, cioè quella che vedrà la rosa più *interamente*.

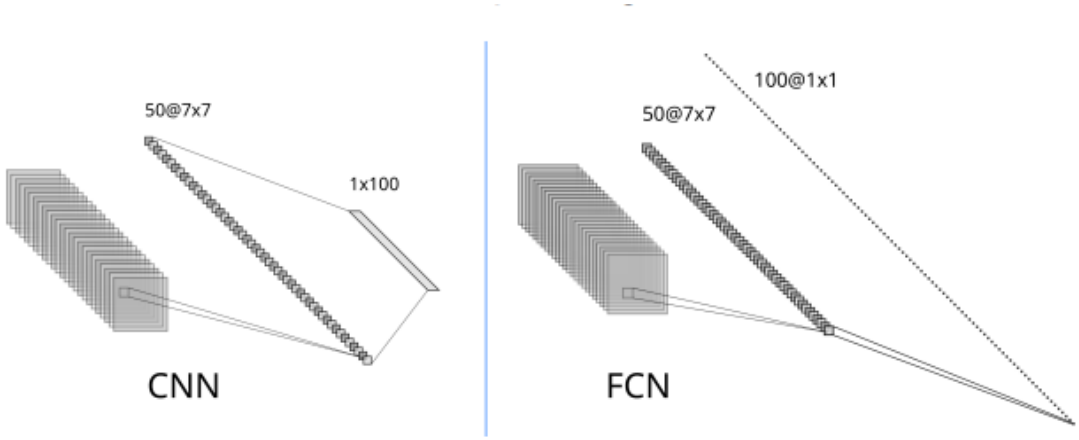


Nelle porzioni di immagine in cui non c'è nulla? (ovvero *background*). Riferendoci all'ultima immagine, se prendiamo una zona in alto a sinistra, una predizione verrà *sempre fatta*, sicuramente con confidenza bassa. Un approccio è creare una classe `background` (predico quando non c'è nulla). Un altro è sfruttare la *bassa confidence*, che scarterò sotto un certo livello.

Altra osservazione: abbiamo risultati differenti in base alla *griglia* che usiamo, ovvero la rete dovrebbe lavorare a *grana fine*. Una soluzione sarebbe lavorare con più reti convoluzionali, che lavorano con griglie diverse. Più *bounding box*, più processamenti. Questo vuol dire più computazione, in quanto dovrei fare anche *post-processamento*.

1.5 Fully Convolutional Networks (FCN)

Reti *completamente convoluzionali*. L'idea è associata ai livelli densi, che non lavorano con immagini variabili, bensì vogliono dimensioni fissi. Soluzione? Rimuovo questi livelli densi, e lascio solo i convoluzionali e di pooling.

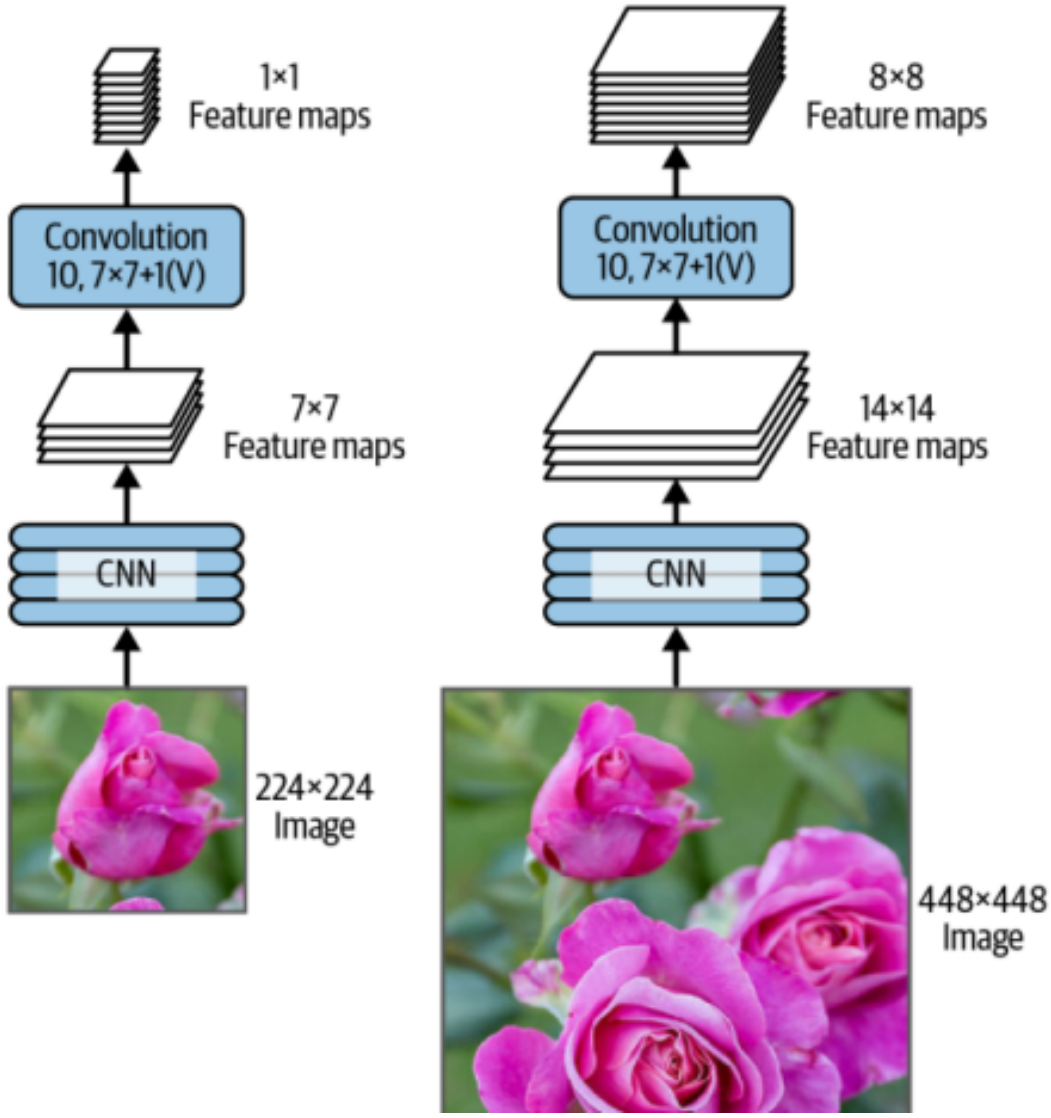


A sinistra, abbiamo un livello connesso con 100 unità, che faranno somma pesata degli input, di dimensione 50x7x7. A destra, mettiamo livello convoluazione, 50 filtri di dimensione 7x7, no padding. Lo applichiamo alle attivazioni del livello precedente, che sono 7x7. Coincidendo, lo applico solo una volta, produce un solo valore. Alla fine producono anche loro 100 elementi, di shape 1x1x100.

Sono equivalenti. Tuttavia, se fornisco *input più grandi* ad entrambi, la rete a sinistra, CNN, smette di funzionare perchè incapacitata a gestire il nuovo input. A destra, la FCN, produrrà 100 feature map di dimensioni 1x1, cioè è in grado di scorrere l'immagine.

Esempio fiori Alleniamo CNN sui fiori, l'output sarà composto da: 5 classi, 4 valori per bounding box, 1 valore di confidenza. Size 224x224. Supponiamo che l'ultimo livello di convoluzione produce feature map 7x7. Se ad **FCN** passiamo immagini 448x448, l'output dell'ultimo livello di convoluzione sarà 14x14. L'ouput sarà 10 feature map, ognuno di 8x8, dove $8 = 14 - 7 + 1$. Partiamo da matrice 14x14, kernel 7x7 scorre, poi passa a pixel 8, pixel 9, cioè scorre di $14 - 7$ volte, più la posizione iniziale.

Questo processo può essere immaginato come se prendessimo la CNN originale e la scorressimo sull'immagine utilizzando 8 passaggi per riga e 8 passaggi per colonna, ma il tutto avviene con un'unica iterazione attraverso la rete. Avrò predizioni per tutte le zone dell'immagine, adattandosi dinamicamente alla size.


1×1
Feature maps
Convolution
10, 7×7+1(V)
7×7
Feature maps
CNN
224×224
Image
8×8
Feature maps
Convolution
10, 7×7+1(V)
14×14
Feature maps
CNN
448×448
Image