Lez26_Autoencoders

January 9, 2024

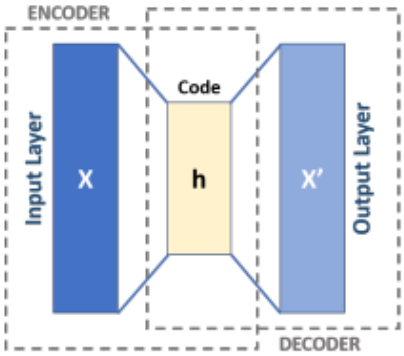
1 Autoencoders

E' una rete neurale che viene addestrata per produrre in in uscita il proprio input, come se fosse una funzione identità. Non ci sono esempi dell'output, in quanto l'uscita è proprio l'input stesso.

Abbiamo rete che riceve in input un vettore X, ed in uscita produce X' che deve essere quanto più possibile simile al vettore X. La cosa è interessante se aggiungo nel mezzo un hidden layer, magari con meno unità rispetto ad X. Queste unità del livello nascosto h devono contenere quante più informazioni possibili per ricostruire il vettore di partenza. Deve codificare in maniera compatta il vettore di input, ovvero una *compressione* dell'input. E' difficile ottenere $X = X$, perchè comunque con la compressione perdo per forza delle informazioni. Se riuscissi ad addestrare tale rete, scopro che per ogni vettore X, il vettore h ci fornisce una visione compatta, estraendo da X le feature più significative, magari da usare su un altro problema di ML.

Two main blocks:

- Encoder: computes $\boldsymbol{h} = f(\boldsymbol{x})$
- Decoder: reconstructs $\boldsymbol{x}' = g(\boldsymbol{h})$



Parliamo di *autoencoder* come la composizione di *encoder* e *decoder*.

Un esempio pratico è: I satelliti inviano foto alla Terra, tante immagini ma poca banda. Allora comprimo le immagine, ma come? Usando algoritmi di compressione come *jpeg*, però le immagini satellitari sono più complesse, con più canali e più frequenze (anche non visibili). Per la compressione uno quindi un *autoencoder*, per ottenere versione compatta, e poi addestro modello per ricostruire immagine in uscita. La versione compatta deve comunque essere abbastanza ricca di informazioni per una corretta ricostruzione. Quindi la prima parte dell'autoencoder va fatta sul satellite, trasferisco le info del primo livello nascosto sulla Terra, e poi dalla Terra ricostruisco il tutto.

Un vincolo classico è sul numero di unità del livello nascosto, andando a complicare (e rendere più interessante) questo processo.

Altre casistiche sono:

- *Riduzione delle dimensioni*: Il codice è costretto ad avere una dimensionalità molto inferiore rispetto all'input. Ci interessa soprattutto la *rappresentazione "compressa"* e latente. L'encoder e il decoder potrebbero essere eseguiti su host diversi, ad esempio comprimendo le immagini satellitari prima della trasmissione.
- *Riduzione del rumore*: L'autoencoder è addestrato per ricostruire l'input originale x, dato una versione rumorosa di x. Ci interessa principalmente l'*output della ricostruzione*.
- *Rilevamento delle anomalie*: Osserviamo l'errore di ricostruzione; un errore elevato potrebbe corrispondere a un input anomalo. Ci interessa principalmente l'*errore di ricostruzione* (ovvero la differenza tra X ed X'). Se facciamo un elettrocardiogramma, mediante autoencoder (addestrato su elettrocardiogrammi di persone in salute), e creerà coding. Addestrato, se propongo altre curve di persone in salute, queste si assomiglieranno, e quindi dopo il coding ricostruisce la curva. Se la curva è molto diversa (la persona sta male), il coding verrà comunque costruita, ma l'errore nella ricostruzione sarà maggiore, in quanto il tipo di dato sarà sensibilmente diverso.

1.1 Costruzione di autoencoder

Composto da encoder e decoder, reti neurali feed-forward con livelli nascosti. L'ultimo livello di uscita del decoder (e quindi dell'autoencoder) deve avere stessa dimensione del primo livello dell'encoder (primo dell'autoencoder), nel mezzo faccio ciò che voglio. La funzione Loss da minimizzare deve tener conto del confronto tra vettore di uscita e quello di entrata. Una possibilità è l'errore quadratico tra l'uscita e l'ingresso, ovvero:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{x}') = ||\boldsymbol{x} - \boldsymbol{x}'||_2^2$$

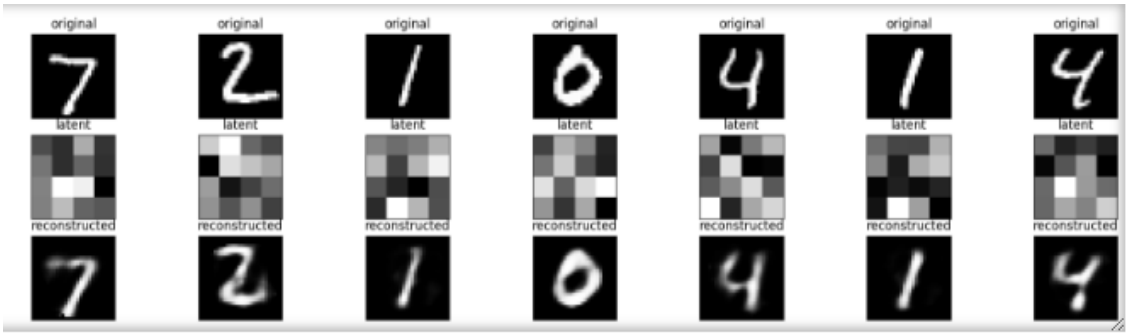
1.2 Esempio notebook: autoencoder_mnist.ipynb

Qui useremo la **binary cross-entropy** come **reconstruction loss**. Pensiamo alla ricostruzione di un pixel come al calcolo della probabilità che il pixel sia nero. Successivamente, confrontiamo la probabilità di output con il valore reale fornito dall'immagine di input. Ricordiamo che la densità di un pixel va da 0 a 255, per questo dobbiamo normalizzarla tra 0 e 1 (cosa da fare anche nei progetti).

```
class Autoencoder(Model):
  def __init__(self, latent_dim, input_shape):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.shape = input_shape
    self.encoder = tf.keras.Sequential([ #definisco encoder
      layers.Flatten(),    #livello che prende una matrice/tensore/immagine e manda in output
                           #un tensore monodimensione, è pre processamento, appiattisce
#(mi da vettore).
      layers.Dense(latent_dim, activation='relu'), #direttamente l'input arriva ad encoder,
```

```
        #che ha unico livello nascosto.
    ])
    self.decoder = tf.keras.Sequential([ #definisco decoder
      layers.Dense(tf.math.reduce_prod(input_shape), activation='sigmoid'),
        #livello uscita calcola
        #dinamicamente 28x28, uso sigmoid perchè produto pixel tra 0 e 1.
      layers.Reshape(input_shape) # fa contrario del flatten, prende vettore e lo risistema
        #sullo shape dato, è post-processamento. Da vettore a immagine.
    ])

  def call(self, x):  #flusso di esecuzione, chiamo prima encoder, poi decoder,
#poi output di quest ultimo.
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded


shape = X_test.shape[1:]
latent_img=(4,4)  #qui usiamo size piu piccole di 28x28
latent_dim = latent_img[0]*latent_img[1]
autoencoder = Autoencoder(latent_dim, shape)

autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())


autoencoder.fit(X_train, X_train, #Qui passiamo due volte X_train, prima era X_train e
#Y_train, perchè ora non siamo più supervisionati.
                epochs=20,
                shuffle=True,
                validation_data=(X_test, X_test), #uso test set come validation set,
                #ma è una cosa non fare mai!
                callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)])
```

A riga 60 del file in esame, vediamo:



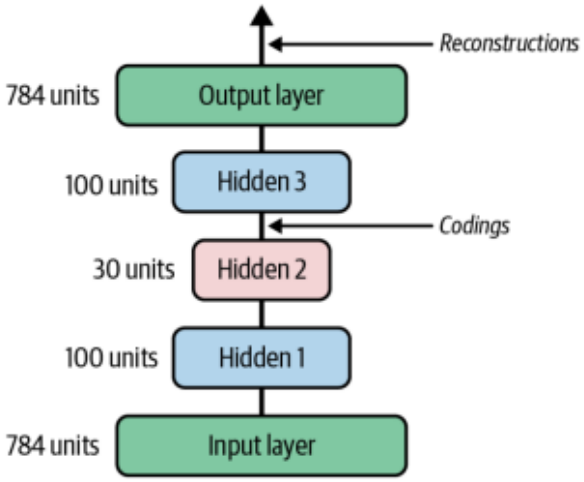
Ovvero numero originale (*original*), codifica (*latent*), e ricostruzione (*reconstructed*). Siamo arrivati a 9 pixel, partendo da più di 700 pixel! Ovviamente la fedeltà dipende da quanto sto codificando.

2 Deep autoencoder

La profondità permette di avere dei coder più complessi, oltre che l'addestramento dell'autoencoder con meno dati. Il training diventa più complicato. Un autoencoder troppo potente potrebbe essere inutile però:

In un caso estremo, un autoencoder con molti strati nascosti e unità potrebbe imparare una rappresentazione perfetta dei dati di addestramento. Ad esempio, se ogni input è codificato come un singolo numero intero (x_0 viene mappato a 0, x_1 a 1 e così via) e il decoder apprende la mappatura inversa, potrebbe sembrare funzionare perfettamente nel set di addestramento. Tuttavia, questo modello probabilmente sarebbe completamente inutile di fronte a dati nuovi o non visti. La sua dipendenza da una corrispondenza perfetta uno a uno nei dati di addestramento non consentirebbe una buona generalizzazione a informazioni nuove o variazioni al di là di ciò su cui è stato addestrato.

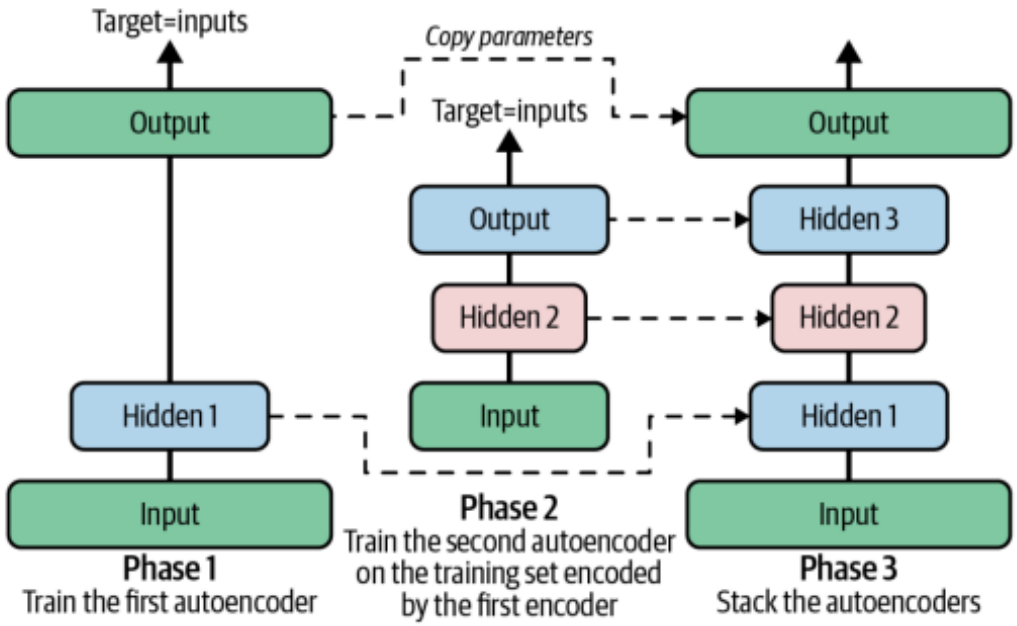
While not necessary, deep autoencoders usually have a symmetrical architecture with a central hidden layer (coding layer)



2.1 Tying Weights - duplicazione pesi

Data la struttura vista sopra, il decoder deve fare l'inverso. Allora i pesi del livello 1 li metto anche a livello 3, a patto che la matrice sia trasposta (perchè il mapping è inverso). Riduce anche overfitting.

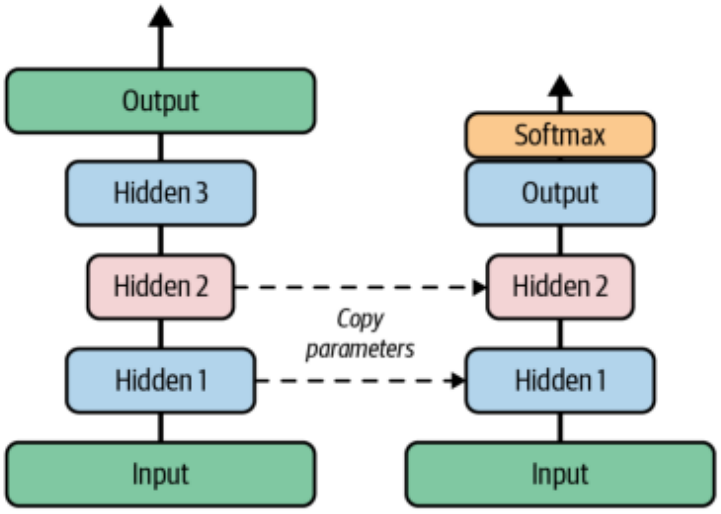
2.2 Training Stacked



Prima addestro autoencoder con un solo livello nascosto (nella foto, a sinistra). Dopo inserisco altro livello nascosto, addestrato da *hidden 1*. Cioè addestro un autoencoder prima coi livelli esterni, poi quelli interni e cosi via.

2.3 Unsupervised Pretraining

Vogliamo associare gli edifici alle facoltà di Tor Vergata. Su internet, difficilmente troveremmo le etichette. I dati sono facili. Oppure gli animali, facili le foto, però le etichette (cane, gatto, ...) sono meno scontate. Se supervisionassi sui pochi dati etichettati, avrò overfitting. L'idea è addestrare su tutti i dati l'*autoencoder*, che impara a costruire nei suoi primi livelli di encoding, delle feature per rappresentare l'immagine data. Tali feature possono essere utili per riconoscere qualsiasi immagine di animale data in input. Riuso questi primi livelli e dopo ci metto uno o più livelli per la classificazione. Se ho pochi dati etichettati, potrei congelare pesi di questi livelli (evitando overfitting). L'autoeconder riconosce le feature, e poi sfrutto i pochi esempi per mappare le feature alle varie classi.



- To avoid overfitting, we may freeze the lower layers when training the final model

2.4 Autoencoders Convoluzionali

Utili per lavorare con le immagini. La particolarità è nel decoder, che deve ricostruire l'immagine originale, e si usa un livello *di convoluzione trasposta*, il quale serve per aumentare la dimensione delle immagini in input (invece di diminuirle come già visto in altre lezioni). Lo *stride* sarà un valore < 1 (di quanti pixel mi sposto), quindi un pixel verrà usato più di una volta per la ricostruzione. Le reti convoluzionali sono più costose da addestrare.

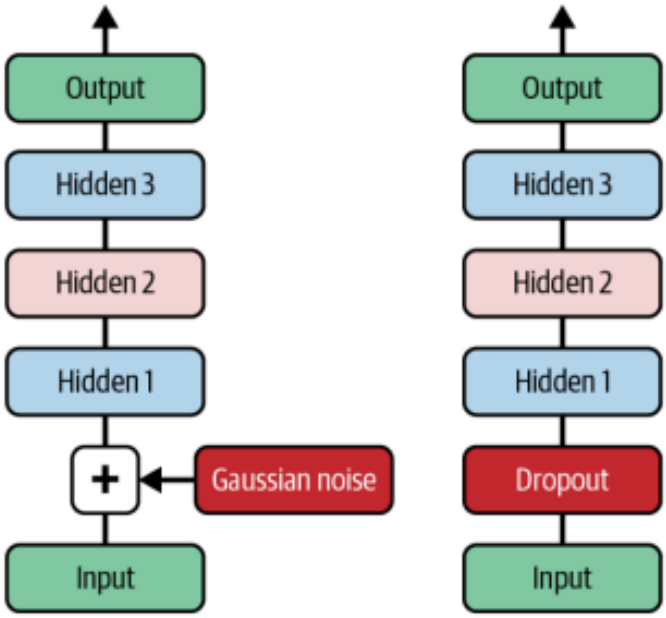
Finora abbiamo considerato **autoencoder sottocompressi/undercomplete**, dove la dimensionalità del codice deve essere più piccola della dimensionalità dell'input.

- L'Autoencoder non può semplicemente inoltrare il suo input all'output
- Deve imparare caratteristiche rilevanti

Ci sono situazioni in cui usiamo altri tipi di vincoli, permettendo al livello di codifica di essere grande quanto l'input, o addirittura più grande (autoencoder sovracompresso).

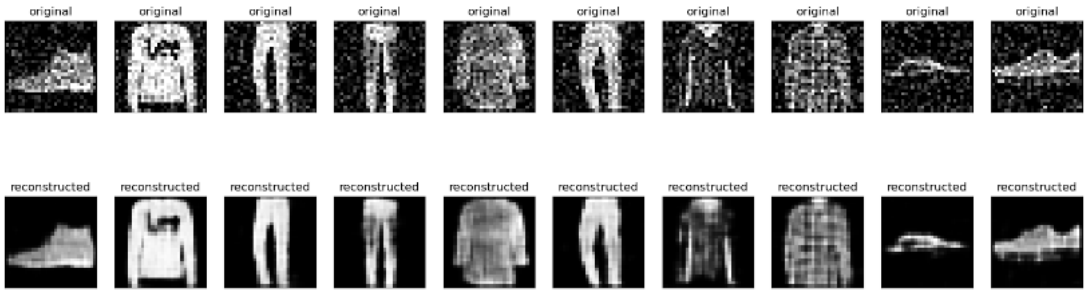
2.5 Denoising Autoencoders DAE

Aggiungiamo rumore agli input, con l'autoencoder che cerca di ricostruire l'input originale privo di rumore. Il rumore può corrispondere a puro rumore gaussiano aggiunto agli input. In alternativa, il rumore può corrispondere agli input messi off casualmente (ossia, dropout).



L'autoencoder deve riuscire a coprire pixel mancanti. L'autoencoder, se ben addestrato, ricostruirà un'immagine simile all'originale quanto più priva di rumore, e sarà il confronto con la foto originale il metro di valutazione.

Prima fornivamo sia per l'input che per l'ouput `X_train`, ora forniamo in input le immagini con rumore, e ci aspettiamo in output l'immagine senza rumore. Ciò che ci aspettiamo è:



2.6 Sparse Autoencoders

Voglio alcuni neuroni sparsi a 0, ovvero feature non significative, e mi interesso solo a quelle con attivazioni non nulle. Il tutto avendo codifica sparsa, usabile anche per altri addestramenti. Nel livello di *coding* uso sigmoid tra 0 e 1, e poi applico la *regolarizzazione L1* ai livelli di attivazione, **non ai pesi**.