Andrea Augello
Department of Engineering, University of Palermo, Italy

Deep learning, Autoencoder

Università degli Studi di Palermo

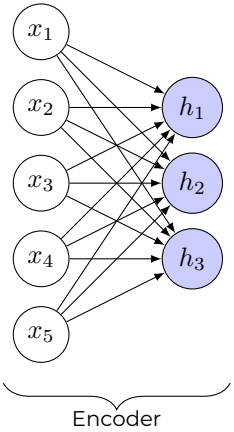
Autoencoder

Autoencoder

- Un autoencoder è una rete neurale che cerca di apprendere una rappresentazione (codifica) di un insieme di dati in modo che possa essere ricostruita (decodifica) il più fedelmente possibile.
- L'autoencoder è composto da due parti:
  - codificatore: trasforma l'input in una rappresentazione interna;
  - decodificatore: trasforma la rappresentazione interna in un output.

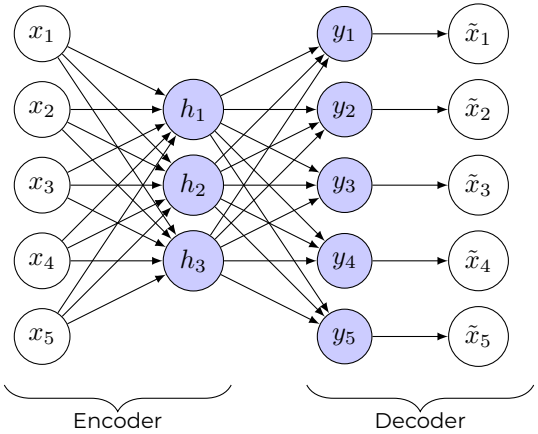
Autoencoder

L'autoencoder più semplice è una rete neurale composta da due soli strati:



Autoencoder

L'autoencoder più semplice è una rete neurale composta da due soli strati:



Implementazione in PyTorch

L'architettura in se non è difficile da implementare, la sintassi sarebbe la stessa di una rete neurale qualsiasi.

```
class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        #...

    def forward(self, x):
        x = self.encoder(x)
        x = self.activation(x)
        x = self.decoder(x)
        return x
```

Come funziona l'addestramento?

Per addestrare l'autoencoder utilizziamo come funzione di costo il Mean Squared Error (MSE) tra l'input e l'output. Questo è un addestramento non supervisionato perché non abbiamo bisogno di etichette per addestrare la rete.

Essendo un addestramento non supervisionato, nel calcolo della loss non possiamo utilizzare le etichette. Useremo quindi l'input sia come input che come target: `loss = criterion(output, x)`.

Il nostro loop di addestramento restituirà anche la lista delle loss per poterle visualizzare.

Prova pratica

Proviamo il nostro Autoencoder su un dataset

Il dataset che utilizzeremo è il dataset Iris che contiene 150 esempi di 3 specie di fiori. Per ogni esempio abbiamo 4 attributi: lunghezza e larghezza del sepalo e del petalo. Inoltre ad ogni campione è associata una etichetta che indica la specie di appartenenza (Setosa, Versicolour, and Virginica).

Iris Versicolor **Iris Setosa** **Iris Virginica**

Carichiamo il dataset in un tensore

```
import torch
from sklearn.datasets import load_iris
import numpy as np

def load_torch_iris():
    dataset = load_iris()
    X = dataset.data
    y = dataset.target
    full_dataset = np.concatenate((X, y.reshape(-1,1)), axis=1)
    return torch.tensor(full_dataset, dtype=torch.float32)
```

Visualizzare il dataset

Essendo un dataset a 4 dimensioni (più label) non possiamo visualizzarlo direttamente. Scegliamo quindi due dimensioni e visualizziamo le label di ogni esempio con un colore diverso.

```
def plot_2d(X, f1=0, f2=1):
    """Plot a 2D representation of a dataset
    :X: dataset
    :f1: index of first feature
    :f2: index of second feature
    """
    plt.scatter(X[:, f1], X[:, f2], c=X[:, -1], edgecolors='black',
        cmap='viridis')
    plt.show()
```

Visualizzare l'errore di ricostruzione

Per visualizzare l'errore di ricostruzione possiamo utilizzare un contour plot. In particolare, tricotourf ci permette di visualizzare le curve di livello di una funzione interpolando i valori mancanti.
Come prima, limitiamo la visualizzazione a due dimensioni.

```
def plot_mse(X, error, labels, f1=0, f2=1):
    X = X.detach().numpy()
    error = error.detach().numpy()
    #show error levels
    plt.tricontourf(X[:, f1], X[:, f2], error)
    plt.colorbar() #show colorbar
    #same as plot_2d
    plt.scatter(X[:, f1], X[:, f2], c=labels, edgecolors='black',
        cmap='viridis')
    plt.show()
```

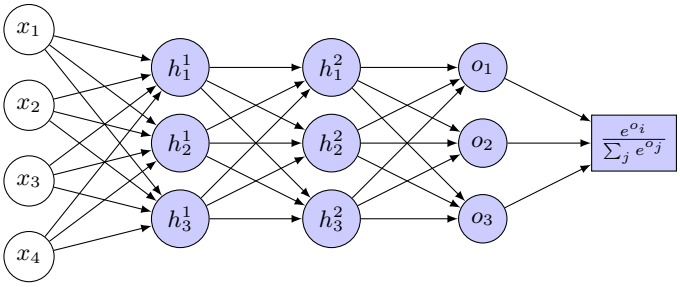
Supporto per la classificazione

Supporto per la classificazione

Un autoencoder può essere utilizzato per estrarre una rappresentazione interna dei dati che sia utile per un problema di classificazione.

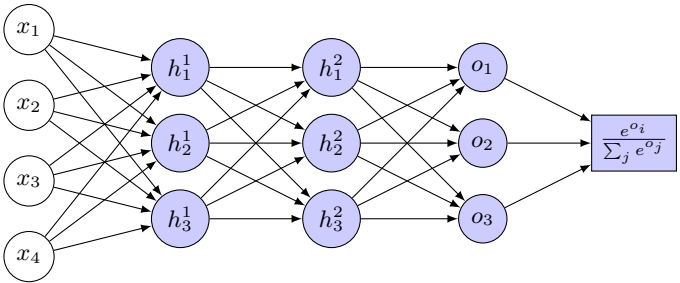
Proviamo quindi a costruire un autoencoder che apprenda una rappresentazione interna dei dati che sia utile per un problema di classificazione sul dataset Iris.

Architettura

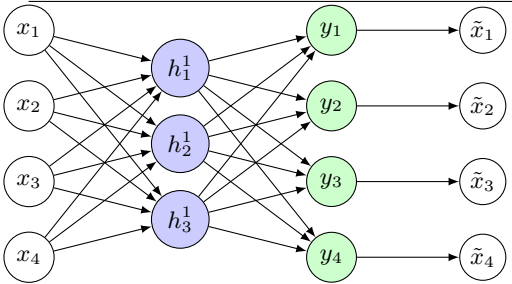


Utilizziamo una rete con tre strati.

Architettura



Utilizziamo una rete con tre strati.



Il primo strato verrà addestrato come autoencoder.

Implementazione in PyTorch — Architettura di base

Per prima cosa proviamo a definire l'architettura della rete senza considerare l'autoencoder.

```
class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        #...

    def forward(self, x):
        #...

    def classify(self, x):
        return torch.argmax(self(x), dim=1)
```

Implementazione in PyTorch — Architettura di base

Per l’addestramento utilizziamo il solito codice, con la cross entropy come funzione di costo. Restituiamo inoltre la lista delle loss per poterle visualizzare.

Valutiamo come si comporta il modello

```
def iris_classifier(pretrain=True):
    iris = dataset.load_torch_iris()
    iris = iris[torch.randperm(iris.size()[0])]
    iris_train, iris_test = iris[:100], iris[100:]
    x = iris_train[:, :-1]
    y = iris_train[:, -1].long()
    classifier = nets.Classifier()
    loss = classifier.train_classifier(x, y)
    plt.plot(loss)
    plt.show()
    x_test = iris_test[:, :-1]
    y_test = iris_test[:, -1].long()
    utils.confusion_matrix(y_test, classifier.classify(x_test))
```

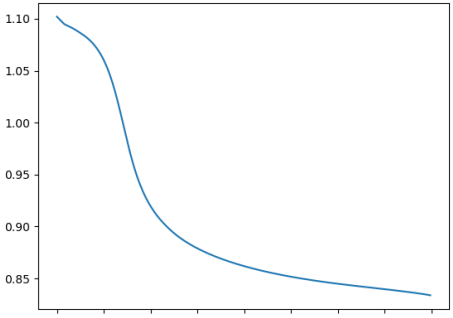
La matrice di confusione

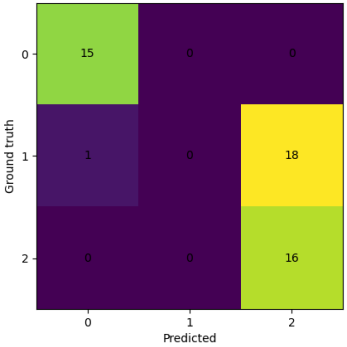
```
def confusion_matrix(y, yhat):
    classes = np.unique(y)
    n_classes = len(classes)
    cm = np.zeros((n_classes, n_classes))
    for i, c in enumerate(classes):
        for j, c2 in enumerate(classes):
            cm[i, j] = ((yhat == c2) * (y
                == c)).sum()
            # add labels
            plt.text(j, i, int(cm[i, j]),
                ha="center", va="center")
    plt.imshow(cm)
    plt.xticks(range(n_classes), classes)
    plt.yticks(range(n_classes), classes)
    plt.xlabel("Predicted")
    plt.ylabel("Ground truth")
    plt.show()
```

Una matrice di confusione è una matrice che mostra le predizioni del modello rispetto alle etichette reali.

È utile per capire quali classi vengono confuse tra loro, specialmente quando abbiamo più di due classi.

I risultati lasciano a desiderare





Implementazione in PyTorch — Pretraining

Il modo più semplice per implementare il pretraining è quello di definire classi distinte per l'encoder, il decoder, e per il classificatore.

Implementazione in PyTorch — Pretraining

Il modo più semplice per implementare il pretraining è quello di definire classi distinte per l'encoder, il decoder, e per il classificatore.

Nel nostro caso non sarebbe indispensabile, vista la semplicità del modello, ma è comunque una buona pratica.

Implementazione in PyTorch — Pretraining

Il modo più semplice per implementare il pretraining è quello di definire classi distinte per l'encoder, il decoder, e per il classificatore.

Nel nostro caso non sarebbe indispensabile, vista la semplicità del modello, ma è comunque una buona pratica.

```
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        #...

    def forward(self, x):
        x = self.fc1(x)
        return self.activation(x)
```

Implementazione in PyTorch — Pretraining

Il modo più semplice per implementare il pretraining è quello di definire classi distinte per l'encoder, il decoder, e per il classificatore.

Nel nostro caso non sarebbe indispensabile, vista la semplicità del modello, ma è comunque una buona pratica.

```
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        #...

    def forward(self, x):
        x = self.fc1(x)
        return self.activation(x)
```

```
class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        #...

    def forward(self, x):
        x = self.fc1(x)
        return x
```

Implementazione in PyTorch — Pretraining

```
class AEClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        #...

    def forward(self, x):
        #...

    def autoencoder(self, x):
        #...

    def classify(self, x):
        return torch.argmax(self(x), dim=1)
```

Come addestrare questa rete?

- Per prima cosa preaddestriamo l'encoder.
  - Per fare ciò utilizziamo il training set senza le etichette.
  - Utilizziamo la MSE come funzione di costo.
  - Servirà addestrare sia l'encoder che il decoder.

Come addestrare questa rete?

- Per prima cosa preaddestriamo l'encoder.
  - Per fare ciò utilizziamo il training set senza le etichette.
  - Utilizziamo la MSE come funzione di costo.
  - Servirà addestrare sia l'encoder che il decoder.
- Successivamente addestriamo il classificatore.
  - Utilizziamo il training set con le etichette.
  - Utilizziamo la cross entropy come funzione di costo.
  - I pesi dell'encoder non dovranno essere modificati.

Addestramento del classificatore e pretraining dell'autoencoder

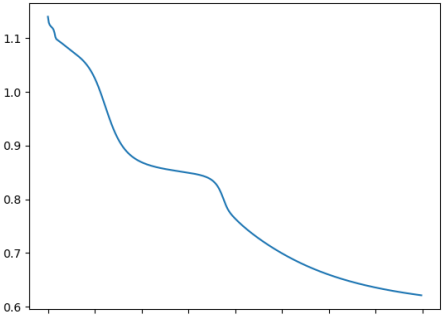
Addestriamo solo l'autoencoder

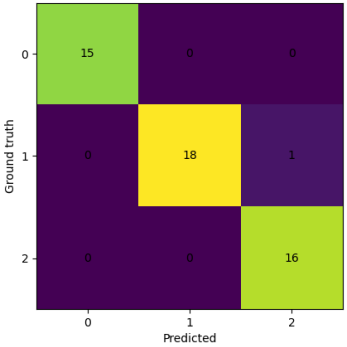
```
    def pretrain_autoencoder(self, x, lr=0.01, epochs=400):
        #...
```

Congeliamo i pesi dell'encoder prima di addestrare il classificatore

```
    def train_classifier(self, x, y, lr=0.1, epochs=400):
        self.pretrain_autoencoder(x)
        for param in self.encoder.parameters():
            param.requires_grad = False
        optimizer = torch.optim.SGD(filter(lambda p: p.requires_grad,
            self.parameters()), lr=lr)
        #...
```

La rete ha un comportamento migliore





Prossimi passi

Prossimi passi

- Vedere cosa succede ai pesi se non congeliamo l'encoder prima di addestrare il classificatore.
- Provare architetture di autoencoder più complesse e confrontare il MSE ottenuto nella ricostruzione dell'input.
  - Ad esempio, provare ad aggiungere un ulteriore strato all'encoder e al decoder.
  - Il primo layer dell'encoder dovrà avere dimensione maggiore rispetto all'input.
  - L'ultimo layer dovrà avere dimensione minore dell'input, altrimenti si rischia di apprendere una funzione identità.