Reti Neurali per il riconoscimento di immagini

PCTO Addestramento di Reti Neurali con Linguaggio Python

Laura Nenzi, Gloria Pietropolli, Gaia Saveri

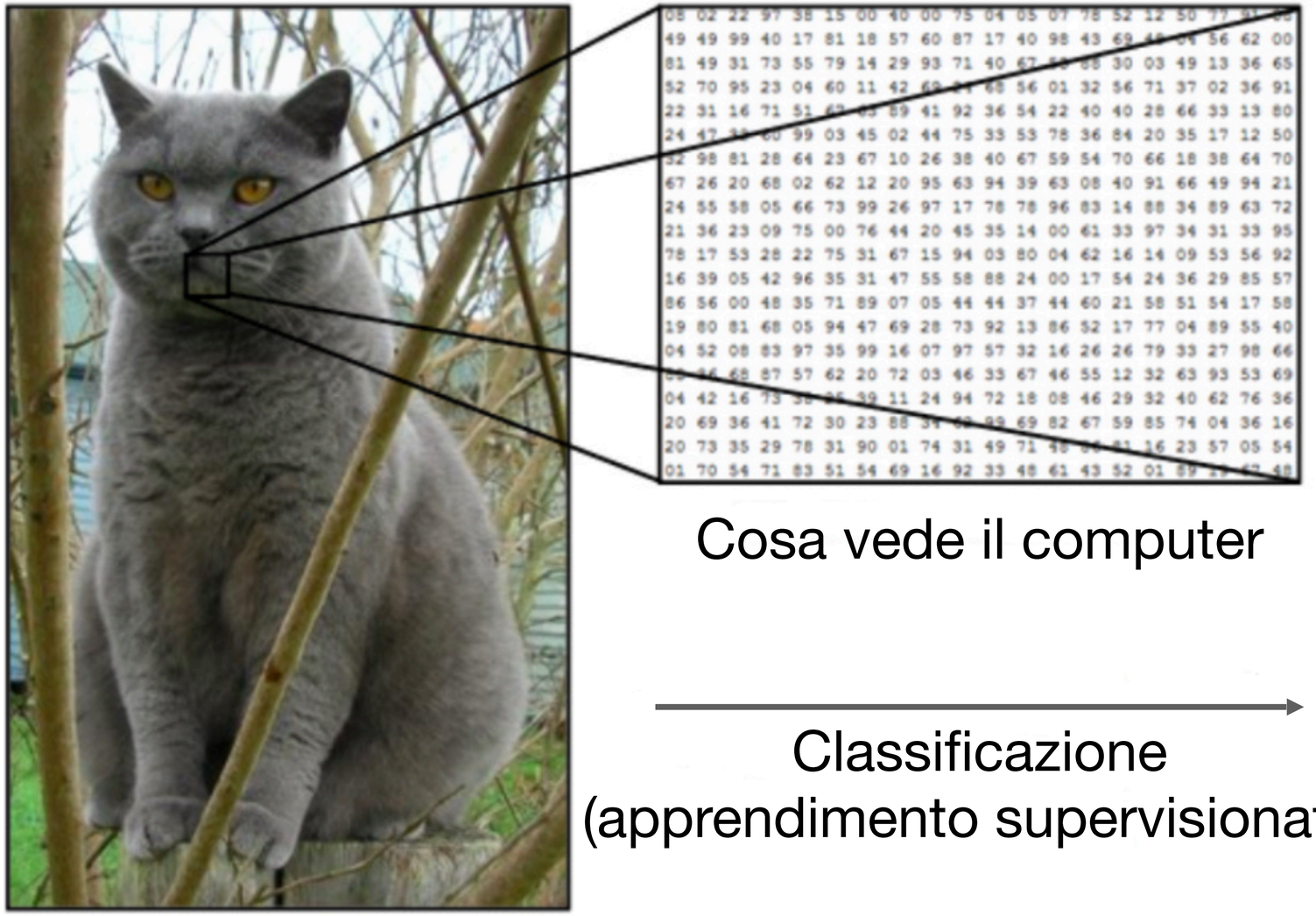
LICEO SCIENTIFICO GALILEO GALILEI

1924 2024

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Reti Neurali per il riconoscimento di immagini

Classificare immagini utilizzando tecniche di Deep Learning



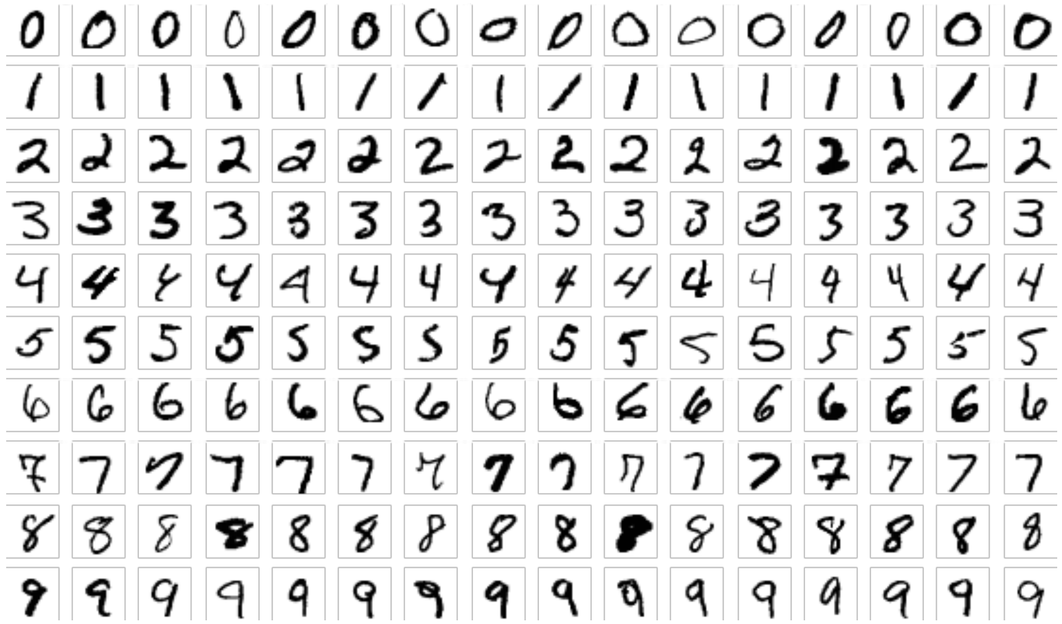
Cosa vede il computer

Classificazione
(apprendimento supervisionato)

85% gatto
15% cane

Ma quindi.. un MLP può fare tutto? Mettiamolo alla prova

Usiamo un dataset ancora più difficile



MNIST Dataset:

- Immagini delle 9 cifre scritte da persone diverse
- Obiettivo: creare un modello in grado di classificare correttamente la cifra visualizzata
- Dataset disponibile su tensorflow

Caricamento e preprocessing di MNIST dataset

MLP Per la classificazione

Facciamo un file mlp_mnist.py

```
6   # Caricamento e preprocessamento del dataset MNIST
7   (x_train_mnist, y_train_mnist), (x_test_mnist, y_test_mnist) = tf.keras.datasets.mnist.load_data()
8
9   # One-hot encoding delle etichette
10  y_train_mnist_enc = tf.keras.utils.to_categorical(y_train_mnist)
11  y_test_mnist_enc = tf.keras.utils.to_categorical(y_test_mnist)
12
13  # Normalizzazione dei pixel delle immagini
14  x_train_mnist_norm = x_train_mnist.astype(np.float32) / 255.0
15  x_test_mnist_norm = x_test_mnist.astype(np.float32) / 255.0
```

Ricordiamoci sempre di importare le librerie necessarie!!

tf == tensorflow
np == numpy
plt == matplotlib.pyplot

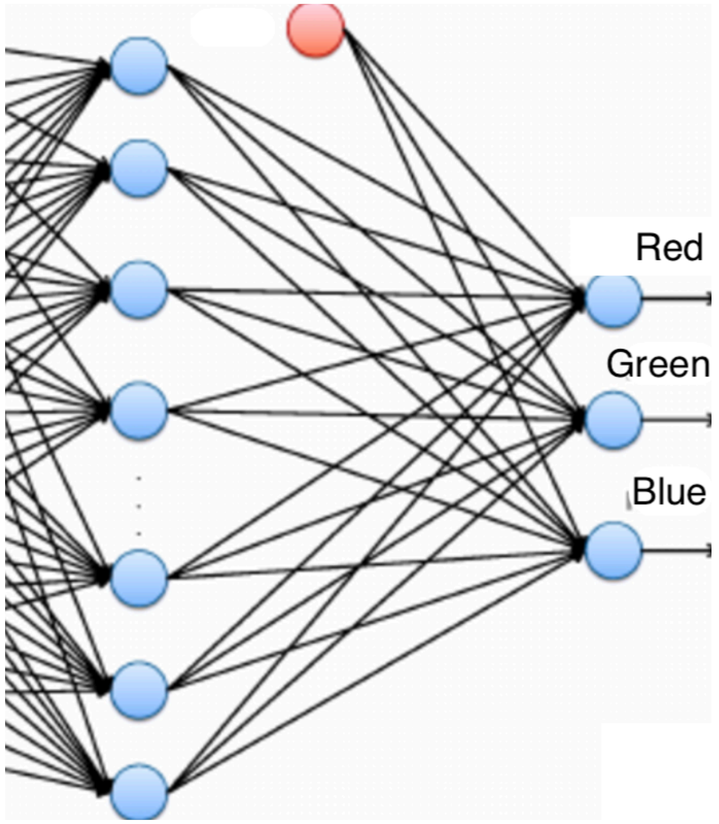
One-hot encoding

Rappresentare variabili categoriche in formato numerico

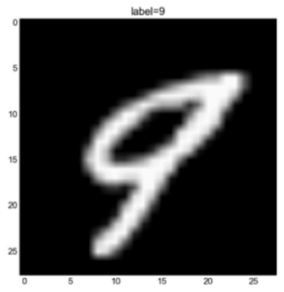
Color
Red
Green
Blue

One-hot encoding

d1	d2	d3
1	0	0
0	1	0
0	0	1



One-hot encoding di MNIST



0,0,0,0,0,0,0,0,0,1

0 1 2 3 4 5 6 7 8 9

Normalizzazione dei dati di input

La normalizzazione è il processo di standardizzazione dei dati di input per migliorare l'efficienza dell'addestramento delle reti neurali.

Perché Normalizzare?:
- Riduce le discrepanze di scala tra le caratteristiche, permettendo alla rete di apprendere più velocemente.
- Aiuta a prevenire l'overfitting mantenendo i pesi della rete in un intervallo gestibile.

Normalizzazione nelle Immagini:
- Per le immagini, tipicamente si divide ciascun pixel per 255 (il massimo valore di un pixel in scala di grigi o in RGB), convertendo i dati in un intervallo [0, 1]

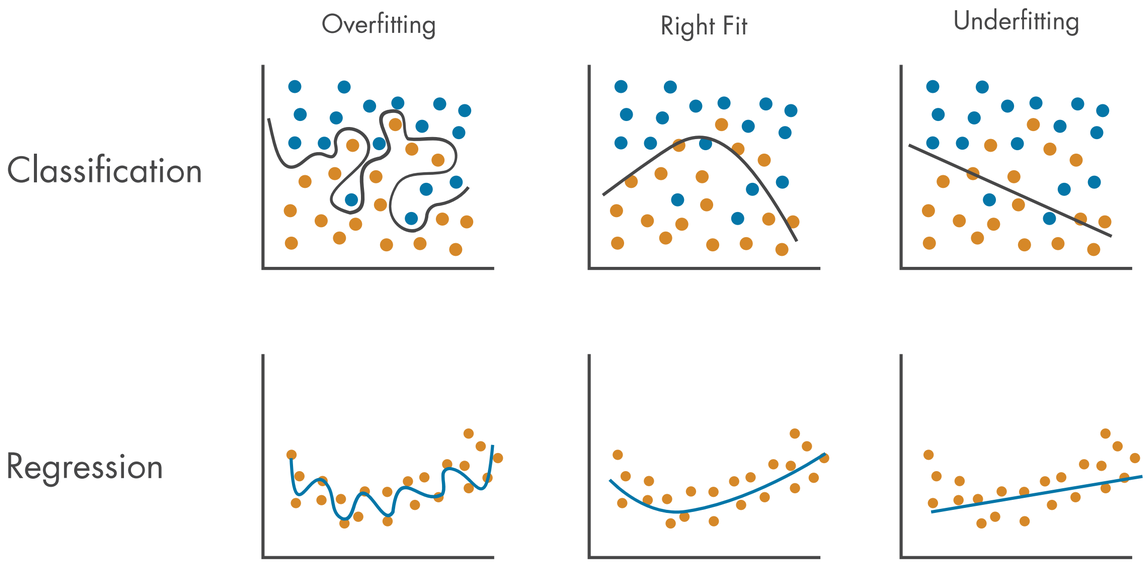
Effetti Visivi:
- Questo processo non altera l'aspetto visivo delle immagini ma rende i dati più omogenei e facilmente interpretabili dalla rete.

Creazione del modello MLP

MLP Per la classificazione

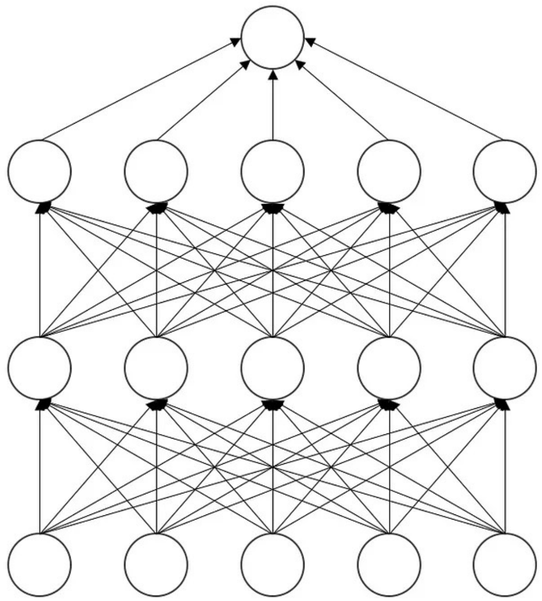
```
17  # Creazione del modello MLP
18  mnist_mlp = tf.keras.models.Sequential([
19      tf.keras.layers.Flatten(input_shape=(28, 28)),  # Flatten per convertire l'immagine in un vettore
20      tf.keras.layers.Dense( units: 128, activation='relu'),  # Primo hidden layer
21      tf.keras.layers.Dropout(0.2),  # Dropout per regolarizzazione
22      tf.keras.layers.Dense( units: 10, activation='softmax')  # Strato di output per classificazione multicl
23  ])
```

Overfitting e Dropout

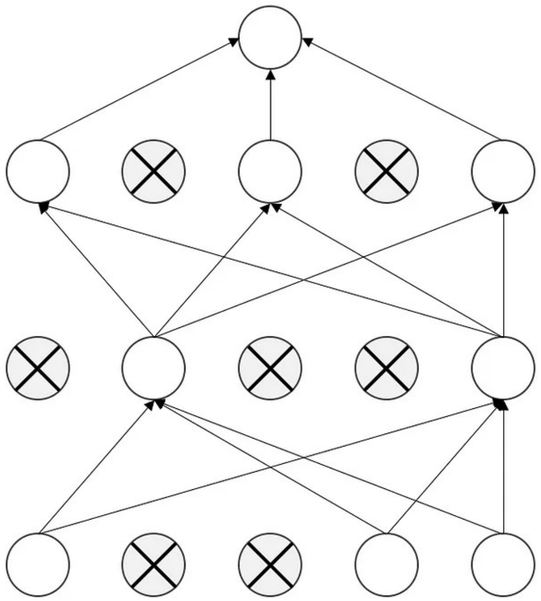


Problema

(Possibile) Soluzione



Standard Neural Net



After applying dropout

Compilazione del modello MLP

MLP Per la classificazione

```
25  # Compilazione del modello
26  lr = 0.001
27  loss_fn = tf.keras.losses.CategoricalCrossentropy()
28  mnist_mlp.compile(
29      optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
30      loss=loss_fn,
31      metrics=['accuracy']
32  )
```

Addestramento del modello MLP

MLP Per la classificazione

```
34  # Addestramento del modello
35  n_epochs = 25
36  mnist_mlp_history = mnist_mlp.fit(
37      x_train_mnist_norm, y_train_mnist_enc,
38      validation_split=0.2,
39      epochs=n_epochs,
40      verbose=1
41  )
```

Valutiamo il modello: grafico dell'errore

MLP Per la classificazione

```
43  # Plot delle perdite di training e validazione
44  train_loss = mnist_mlp_history.history['loss']
45  val_loss = mnist_mlp_history.history['val_loss']
46  epochs = range(1, len(train_loss) + 1)
47
48  plt.figure(figsize=(8, 6))
49  plt.plot( *args: epochs, train_loss, 'r-', label='Training Loss')
50  plt.plot( *args: epochs, val_loss, 'b--', label='Validation Loss')
51  plt.title('Andamento della Loss durante l\'addestramento')
52  plt.xlabel('Epoche')
53  plt.ylabel('Loss')
54  plt.legend()
55  plt.tight_layout()
56  plt.show()
```

Valutiamo il modello: confusion matrix

MLP Per la classificazione

```
62  # Predizioni sul test set
63  y_pred_mnist_prob = mnist_mlp.predict(x_test_mnist_norm)
64  y_pred_mnist = np.argmax(y_pred_mnist_prob, axis=1)
65
66  # Matrice di confusione
67  conf_matrix = confusion_matrix(y_test_mnist, y_pred_mnist)
68  ConfusionMatrixDisplay(conf_matrix, display_labels=np.arange(10)).plot(colorbar=True, cmap='viridis')
69  plt.title("Matrice di Confusione (MNIST)")
70  plt.show()
```

Vorrei fare altri plot…

…però che noia dover ri-addestrare il modello!

Serve la libreria pickle

```
# Salvare le metriche, le etichette vere e predette
results = {
    'history': mnist_mlp_history.history,
    'test_loss': mnist_test_loss,
    'test_accuracy': mnist_test_accuracy,
    'true_labels': y_test_mnist.flatten(),
    'predicted_labels': y_pred_mnist,
    'predicted_probabilities': y_pred_mnist_prob,
    'true_features': x_test_mnist_norm
}

with open(f'mlp_mnist_{n_epochs}_epochs_{lr}_lr.pkl', 'wb') as file:
    pickle.dump(results, file)
```

Analisi dei risultati

Prima cosa: carichiamoli!

Facciamo un file metrics.py

```
# Caricare i dati
n_epochs = 25
lr = 0.001
with open(f'mlp_mnist_{n_epochs}_epochs_{lr}_lr.pkl', 'rb') as file:
    results = pickle.load(file)
```

Analisi dei risultati

Con quale probabilità è stato predetta la classe di un'immagine?

```
# Plot di un'immagine e la sua probabilità predetta
def plot_immagine_prob(prob, img):  1 usage
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(img, cmap='gray')
    y_pred = np.argmax(prob)
    plt.xlabel('{} ({:.2f}%)'.format( *args: int(y_pred), 100 * np.max(prob)))
```

Vediamo se la previsione visivamente ci convince

Quanto è sicuro il modello della classe predetta?

```
# Plot delle probabilità per ogni classe
def plot_prob(prob, y):  1 usage
    plt.grid(False)
    plt.yticks([])
    plt.xticks(np.arange(10))
    prob_bar = plt.bar(np.arange(10), prob, color='grey')
    plt.ylim([0, 1])
    y_pred = np.argmax(prob)
    prob_bar[y].set_color('red')
    prob_bar[y_pred].set_color('green')
```

Analisi dei risultati

Con quale probabilità è stato predetta la classe di un’immagine?

```
plot_idx = np.random.choice(results['predicted_labels'].shape[0], size: 1)
plt.figure(figsize=(6, 3))
plt.subplot( *args: 1, 2, 1)
plot_immagine_prob(
    results['predicted_probabilities'][plot_idx].squeeze(),
    results['true_features'][plot_idx].squeeze()
)
plt.subplot( *args: 1, 2, 2)
plot_prob(
    results['predicted_probabilities'][plot_idx].squeeze(),
    results['true_labels'][plot_idx].squeeze()
)
plt.show()
```

Analisi dei risultati

Con quale probabilità è stato predetta la classe di un'immagine?

```python
# Funzione per calcolare e stampare la precisione per classe
def print_class_accuracy(true_labels, predicted_labels, classes):  1 usage
    accuracy = accuracy_score(true_labels, predicted_labels)
    print(f"Overall accuracy: {accuracy * 100:.2f}%\n")
    print("Accuracy per class:")
    for i, class_name in enumerate(classes):
        class_accuracy = np.mean([true_labels == predicted_labels][0][true_labels == i])
        print(f"{class_name}: {class_accuracy * 100:.2f}%")
```

```python
# Precisione per classe
classes = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']  # MNIST classes
print_class_accuracy(results['true_labels'], results['predicted_labels'], classes)
```

Analisi dei risultati

Metriche durante il training: creiamo una funzione per evitare duplicazione codice

```python
# Funzione per visualizzare il training
def plot_training_history(history):  1 usage
    fig, axes = plt.subplots( nrows: 1,  ncols: 2, figsize=(12, 5))
    train_loss = history['loss']
    train_acc = history['accuracy']
    val_loss = history['val_loss']
    val_acc = history['val_accuracy']
    epochs = range(1, len(train_loss) + 1)

    axes[0].plot(epochs, train_loss, 'r-', label='Training Loss')
    axes[0].plot(epochs, val_loss, 'b--', label='Validation Loss')
    axes[0].set_title('Andamento della Loss durante l\'addestramento')
    axes[0].set_xlabel('Epoche')
    axes[0].set_ylabel('Loss')

    axes[1].plot(epochs, train_acc, 'r-', label='Training Accuracy')
    axes[1].plot(epochs, val_acc, 'b--', label='Validation Accuracy')
    axes[1].set_title('Andamento della Loss durante l\'addestramento')
    axes[1].set_xlabel('Epoche')
    axes[1].set_ylabel('Accuracy')
    plt.legend()
    plt.tight_layout()
    plt.show()
```

```python
# Plot dell'andamento di loss e accuracy durante il training
plot_training_history(results['history'])
```

Analisi dei risultati

Confusion Matrix: creiamo una funzione per evitare duplicazione di codice

sns == seaborn

```python
# Funzione per la matrice di confusione
def plot_confusion_matrix(true_labels, predicted_labels, classes):  1 usage
    cm = confusion_matrix(true_labels, predicted_labels)
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabels=classes)
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()
```

```python
# Matrice di confusione
plot_confusion_matrix(results['true_labels'], results['predicted_labels'], classes)
```