Lez7_TensorFlow_K

November 5, 2023

1 TensorFlow

Libreria per ML focalizzata per DNNs. Supporta addestarmento di reti neurali per vario tipo, ha supporto nativo per addestramento distribuito e su GPU (anche quindi con molteplici nodi o molteplici GPU per aumentare le prestazioni). (se serve esistono versioni per web o dispositivi mobili, utile per inferenza più che per addestramento, si parte da modello già addestrato).

Per installare TensorFlow:

```
pip install tensorflow
conda install tensorflow
```

Nota

E' possibile riscontrare questo *warning*, che ci informa che potremmo ottimizzare le prestazioni di TensorFlow ricompilando il codice sorgente con alcuni flag. E' possibile sopprimere tale messaggio con:

```
#import os
#os.environ['TF_CPP_MIN_LOG_LEVEL'] = '1'
```

Comunque, tale warning è ritornato solo alla prima esecuzione.

```
2023-11-04 18:08:41.728559: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical
operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

2 Keras

Altra libreria opensource per reti neutali. Offre interfaccia Python per costruire modelli per reti neurali, facilita la definizione di una rete neurale, ma ha bisogno di un *backend* per farla girare. Tra i backend disponibili si ha *TensorFlow*. Internamente a TensorFlow, dal 2017 è stata importata la libreria di Keras, ovver `tf.keras`. Non serve installarlo a parte.

2.1 TensorFlow API

API che permettono di definire modelli di reti neurali. 3 interfacce, le prime 2 importate da keras, utilizzeremo quasi solo queste 2: - *Sequential API*: copre la maggior parte dei casi d'uso, sequential fa riferimento al fatto che il modello è una sequenza di livelli che applicano trasformazioni. - *Functional API*: offre più flessibilità, è necessaria quando si vogliono costruire modelli più complessi rispetto alle reti multilivello. - *Core API*: di più basso livello, per fare cose più avanzate, utilizzata molto di rado (voglio sviluppare nuovo tipo di rete neurale che non è soddisfatta dagli altri due livelli).

[58]:
```
#import os
#os.environ['TF_CPP_MIN_LOG_LEVEL'] = '1'


import tensorflow as tf

#La struttura dati su cui lavora è il tensore (array multidimensionale simile a␣
 ↪quelli di numpy). La classe tensor è quella che rappresnta questo tipo di␣
 ↪oggetto.

#Per costruire un tensore basta utilizzare la funzione costant passando i␣
 ↪valori con cui inizializzarlo:
x = tf.constant([[1., 2., 3.],
                 [4., 5., 6.]])
print(x)
#Per stampare basta fare: print(x)
#questo comando mi da la dimensione del tensore print(x.shape)
#mi da il tipo degli elementi nel tensore print(x.dtype)
```

```
tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)
```

[6]:
```
import tensorflow as tf

x = tf.constant([[1., 2., 3.],
                 [4., 5., 6.]])

print(x)
print(x.shape)
print(x.dtype)
```

```
tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)
(2, 3)
<dtype: 'float32'>
```

I più importanti attributi di un tf.Tensor sono la sua forma (shape) e il suo tipo di dati (dtype):

- `Tensor.shape`: ti indica le dimensioni del tensore lungo ciascuno dei suoi assi.
- `Tensor.dtype`: ti indica il tipo di dati di tutti gli elementi presenti nel tensore.

Su questi oggetti si possono fare diverse operazioni (molto simili a quello che si può fare con numpy).

[7]:
```
x + x
```

[7]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```

[8]:
```
5 * x
```

[8]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 5., 10., 15.],
       [20., 25., 30.]], dtype=float32)>
```

[9]:
```
x @ tf.transpose(x)
#La chiocciola rappresenta prodotto tra matrici e vettori.
```

[9]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

[10]:
```
#Posso concatenare un vettore con se stesso o altri tensori.
tf.concat([x, x, x], axis=0)
```

[10]:
```
<tf.Tensor: shape=(6, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.],
       [1., 2., 3.],
       [4., 5., 6.],
       [1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

[11]:
```
#Ho funzioni già implementate che derivano dalle reti neurali (es. softmax).
tf.nn.softmax(x, axis=-1)
```

[11]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.09003057, 0.24472848, 0.66524094],
       [0.09003057, 0.24472848, 0.66524094]], dtype=float32)>
```

[12]:
```
#Sommare tutti gli elementi del vettore:
tf.reduce_sum(x)
```

[12]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=21.0>
```

Nota: Tipicamente, in qualsiasi punto in cui una funzione TensorFlow richiede un tensore come input, la funzione accetterà anche qualsiasi cosa possa essere convertita in un tensore utilizzando tf.convert_to_tensor. Di seguito è riportato un esempio.

[13]:
```
tf.convert_to_tensor([1,2,3])
```

[13]:
```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([1, 2, 3], dtype=int32)>
```

[14]:
```
tf.reduce_sum([1,2,3])
```

[14]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=6>
```

[15]:
```
#Per sapere se l'ambiente che si sta utilizzando usa una GPU:

if tf.config.list_physical_devices('GPU'):
  print("TensorFlow **IS** using the GPU")
else:
  print("TensorFlow **IS NOT** using the GPU")
```

```
TensorFlow **IS NOT** using the GPU
```

2.2 Conversion to NumPy

You can convert a tensor to a NumPy array either using np.array or the tensor.numpy method:

[16]:
```
import numpy as np

np.array(x)
# or
x.numpy()
```

[16]:
```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

2.3 Variables

Il tensore è un oggetto immutabile, quello che viene prodotto è sempre un nuovo tensore che può essere riassegnato alla variabile x. Per fare riferimento a variabili si utilizza la classe Variables, ha uno stato mutabile.

[15]:
```
var = tf.Variable([0.0, 0.0, 0.0]) #inizializzo
```

[16]:
```
var.assign([1, 2, 3]) #assegno valore
```

[16]:
```
<tf.Variable 'UnreadVariable' shape=(3,) dtype=float32, numpy=array([1., 2.,
3.], dtype=float32)>
```

[17]:
```
var.assign_add([1, 1, 1]) #aggiungere un vettore componente per componente
```

[17]:
```
<tf.Variable 'UnreadVariable' shape=(3,) dtype=float32, numpy=array([2., 3.,
4.], dtype=float32)>
```

Una variabile appare e si comporta come un tensore e, infatti, è una struttura dati supportata da un `tf.Tensor`. La maggior parte delle operazioni sui tensori funziona sulle variabili.

2.4 Differenziazione automatica

La discesa del gradiente e gli algoritmi correlati sono un pilastro del machine learning moderno.

Per rendere questo possibile, TensorFlow implementa la differenziazione automatica (autodiff), che utilizza il calcolo differenziale per calcolare i gradienti. Tipicamente, verrà utilizzata per calcolare il gradiente dell'errore o della perdita di un modello rispetto ai suoi pesi.

[17]:
```
x = tf.Variable(1.0)

def f(x):
  y = x**2 + 2*x - 5
  return y
```

[18]:
```
f(x)
#Eseguendo f(x) si ottiene un tensore
#<tf.Tensor: shape=(), dtype=float32, numpy=-2.0>.
```

[18]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=-2.0>
```

La derivata di y è $y' = f'(x) = (2 \cdot x + 2) = 4$. TensorFlow può calcolarlo automaticamente (uno dei motivi per cui si utilizza):

[20]:
```
with tf.GradientTape() as tape:
  y = f(x)

g_x = tape.gradient(y, x)  # g(x) = dy/dx

g_x # g(x) = dy/dx
```

[20]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=4.0>
```

Tutto quello eseguito dentro GradientTape è utilizzato da TensorFlow per creare un grafo computazionale. Viene creata una variabile y basata sulla funzione f passata applicata su x.

Questa definizione permette di utilizzare `tape.gradient` per fare la derivata di y rispetto a x.

Questo esempio semplificato calcolava la derivata rispetto a un singolo scalare (x), ma TensorFlow può calcolare il gradiente rispetto a qualsiasi numero di tensori non scalari contemporaneamente.

2.5 Ottimizzazioni

Per ottimizzare le prestazioni di ciascun blocco di codice si può utilizzare il decoratore:

`@tf.function` Ottimizzarla vuol dire che la prima volta che viene costruito il grafo computazionale ottimizzato, nelle successive invocazioni verrà utilizzato il grafo costruito.

NOTA: compilandolo in un grafo e non eseguendo codice python non stampa le scritte.

Ho utilizzato la funzione su numeri interi, grafo costruito per numeri interi. Se gli passo numeri a virgola mobile deve ricompilare il metodo.

Esistono strumenti di più alto liello per creare astrazioni al di sopra di trensori e variabili.

Esempio Leggermene più complesso: Inizializzo una matrice W 3*2 inizializzata randomicamente, inizializzo `b` come un vettore di 0, definisco poi un vettore `x` di input formato da valori costanti.

[19]:
```
w = tf.Variable(tf.random.normal((3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]

with tf.GradientTape(persistent=True) as tape:
  y = x @ w + b #dentro al blocco Gradient vado a implementare il perceptron
  loss = tf.reduce_mean(y**2)

[dl_dw, dl_db] = tape.gradient(loss, [w, b])
##calcolo il gradiente della loss prima rispetto a w poi rispetto a b,
# tra parentesi indico le variabili rispetto alle quali voglio derivare.
print(dl_db)
```

```
tf.Tensor([-5.01577   -5.2975855], shape=(2,), dtype=float32)
```

In alternativa, possiamo passare un dizionario di variabili:

Poi calcolo il gradiente rispetto a questo dizionario. Mi viene restituito un dizionario contenente i gradienti.

[20]:
```
my_vars = {
    'w': w,
    'b': b
}

grad = tape.gradient(loss, my_vars)
grad['b']
```

[20]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([-5.01577   , -5.2975855],
dtype=float32)>
```

Nota: Per impostazione predefinita, le risorse gestite da un GradientTape vengono rilasciate non appena il metodo `radientTape.gradient` viene chiamato. Per calcolare più gradienti sulla stessa computazione, è possibile creare un gradient tape con `persistent=True`. Ciò consente più chiamate al metodo gradient, poiché le risorse vengono rilasciate quando l'oggetto tape viene raccolto dal garbage collector.

Anche se le variabili sono importanti per la differenziazione, alcune variabili potrebbero non aver bisogno di essere differenziate. È possibile disattivare i gradienti per una variabile impostando `trainable=False` durante la creazione. Un esempio di una variabile che non ha bisogno di gradienti è un contatore di passi di addestramento.

[21]:
```
step_counter = tf.Variable(1, trainable=False)
```

2.6 Gradiente di target non scalari e Jacobiano

Se voglio calcolare il gradiente di qualsiasi uscita y_i rispetto alle variabili x e w, applicare il gradient non mi ritorna ciò che mi aspettavo (calcola il gradiente di ciascuna componente rispetto alle 2, poi i valori vengono sommati).

Per avere il gradiente del vettore y rispetto agli altri vettori utilizzare il **metodo Jacobian**.

[30]:
```
x = tf.linspace(-10.0, 10.0, 5) #vettore 5 elementi
w = tf.Variable([0.0,1.0])      #vettore 2 elementi
with tf.GradientTape(persistent=True) as tape:
  y = (x+w[0]-2*w[1])**3 #costruzione di vettore y
print("- Gradient way, non ritorna ciò che ci aspettavamo!")
print(tape.gradient(y, w).numpy())
print()
print("- Jacobian way")
print(tape.jacobian(y, w).numpy())
```

```
- Gradient way, non ritorna ciò che ci aspettavamo!
[  810. -1620.]

- Jacobian way
[[ 432. -864.]
 [ 147. -294.]
 [  12.  -24.]
 [  27.  -54.]
 [ 192. -384.]]
```

2.7 tf.module e Graoh

Una volta costruito il modello, questo si baserà su variabili "sciolte" python. Il modulo le raggruppa, solitamente risponde all'esigenza di salavare e ricaricare il modello. Se spengo server, non voglio perdere dati, allora salvo il modulo, cioè tutte le variabili, pesi etc… che posso ricaricare quando voglio.

Utile se sto facendp il training di un modello che richiede molto tempo,dopo un tot si spegne il server e bisogna ricominciare da capo. Quindi periodicamente si fa checkpoint, salvando il modulo.

Si crea una nuova classe che estende la classe modulo, definisco una variabile *weight*, il peso. A questo punto definisco l'operazione con la decorazinoe `@tf.function`. E posso salvare quando voglio con tf.save(?)

Ottengo lo stesso risultato, non devo reinizializzare il peso perché già è salvato.

Al di sopra di questo sono stati costruiti la classe level e la classe modello.

Una rete neurale è un insieme di livelli (comprende il loro peso), il livello è un insieme di moduli.

[31]:
```
@tf.function
def my_func(x):
  print('Tracing.\n')
  return tf.reduce_sum(x)
```

La prima volta che si esegue `tf.function`, anche se viene eseguito in Python, cattura un grafo completo e ottimizzato che rappresenta i calcoli TensorFlow eseguiti all'interno della funzione.

[32]:
```
x = tf.constant([1, 2, 3])
my_func(x)
```

```
Tracing.
```

[32]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=6>
```

Nelle chiamate successive, TensorFlow esegue solo il grafo ottimizzato, evitando eventuali passi non correlati a TensorFlow. Di seguito, si nota che `my_func` non stampa il tracciamento poiché la funzione `print` è una funzione Python e non una funzione TensorFlow.

[33]:
```
x = tf.constant([10, 9, 8])
my_func(x)
```

[33]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=27>
```

Un grafico potrebbe non essere riutilizzabile per input con una firma diversa (forma e tipo di dato), pertanto verrà generato un nuovo grafico al posto di quello precedente.

[34]:
```
x = tf.constant([10.0, 9.1, 8.2], dtype=tf.float32)
my_func(x)
```

```
Tracing.
```

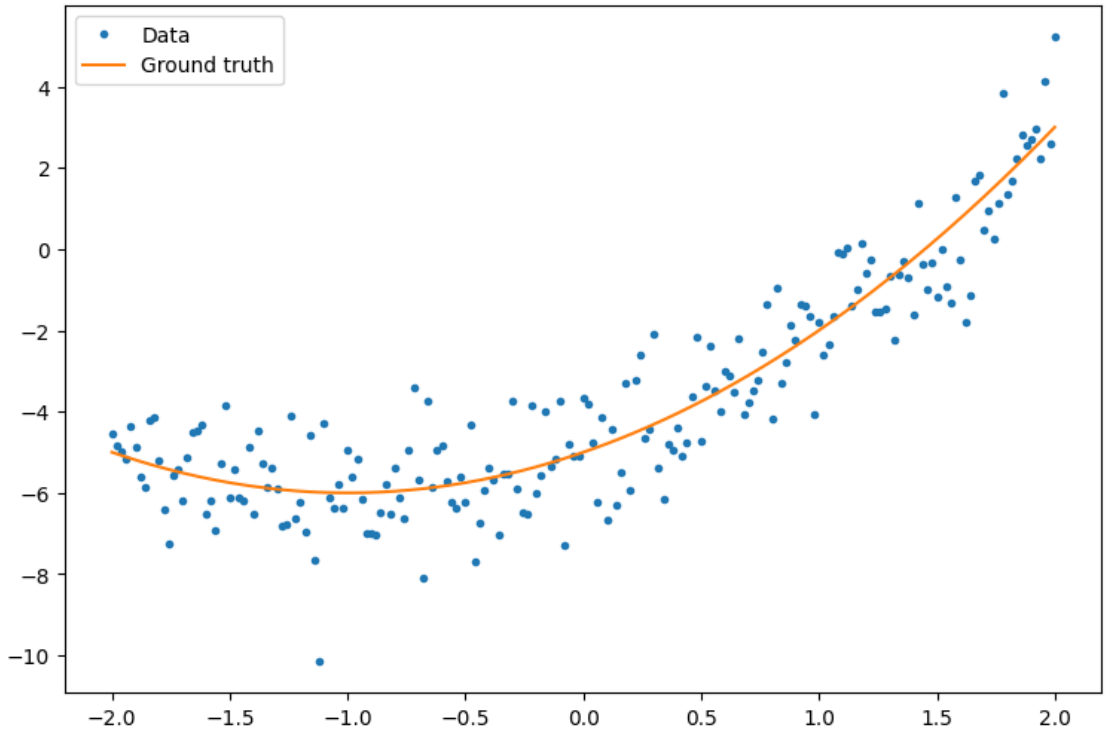
[34]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=27.3>
```

2.7.1 Addestramento di un modello senza usare astrazioni

Genero dei dati, grafico la funzione. Ad ogni valore $f(x)$ viene aggiunto un numero casuale estratto da una normale che fa shiftare il valore del campione (rappresenta il rumore).

Vorrei addestrare un modello per imparare la funzione sottostante a partire dai punti blu. E' un problema di **regressione**. Viene costruito un modulo per risolvere questo problema di regressione. 3 parametri $w1$, $w2$ e b.

[37]:
```
import matplotlib
from matplotlib import pyplot as plt

matplotlib.rcParams['figure.figsize'] = [9, 6]
x = tf.linspace(-2, 2, 201)
x = tf.cast(x, tf.float32)

def f(x):
  y = x**2 + 2*x - 5
  return y

y = f(x) + tf.random.normal(shape=[201])

plt.plot(x.numpy(), y.numpy(), '.', label='Data')
plt.plot(x, f(x), label='Ground truth')
plt.legend();
```

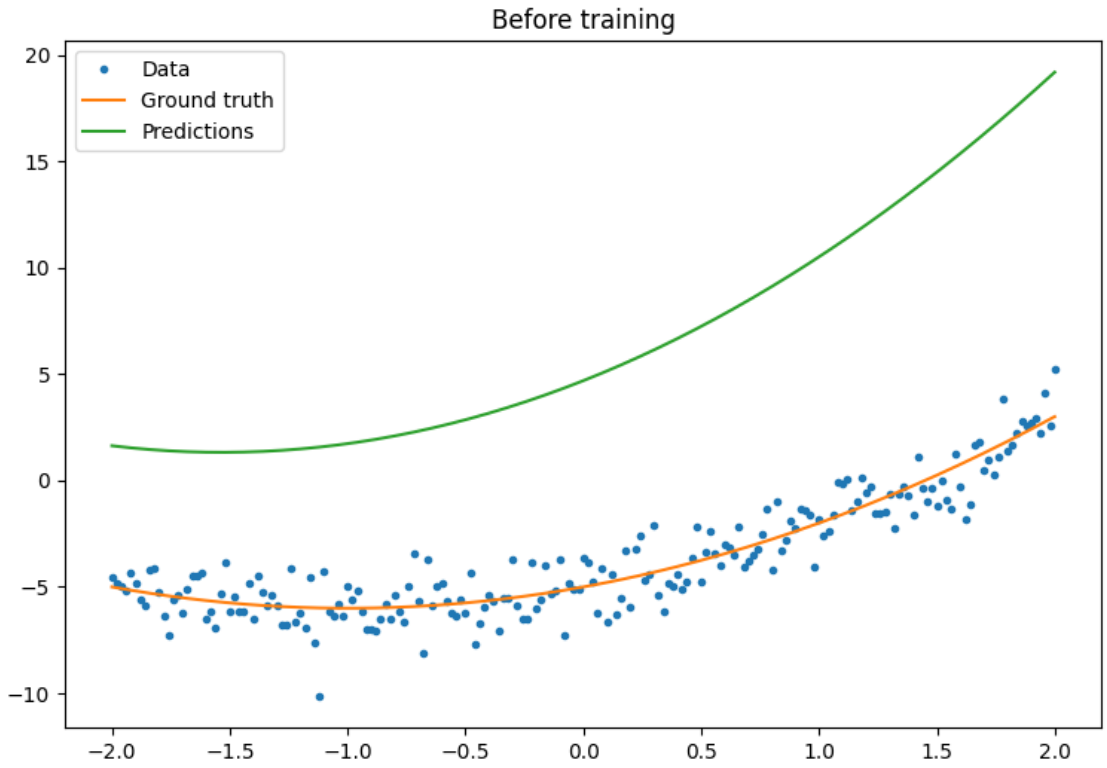


Unico metodo **call** , modello quadratico per fare regressione, parametri inizializzati tramite training. Se li inizializzo in modo casuale le predizioni del modello sono abbastanza fuori da quello che mi aspetto.

[38]:
```
class Model(tf.Module):

  def __init__(self):
    # Randomly generate weight and bias terms
    rand_init = tf.random.uniform(shape=[3], minval=0., maxval=5., seed=22)
    # Initialize model parameters
    self.w_q = tf.Variable(rand_init[0])
    self.w_l = tf.Variable(rand_init[1])
    self.b = tf.Variable(rand_init[2])

  @tf.function
  def __call__(self, x):
    # Quadratic Model : quadratic_weight * x^2 + linear_weight * x + bias
    return self.w_q * (x**2) + self.w_l * x + self.b
```

[40]:
```
#vediamo cosa otteniamo senza training
quad_model = Model()

def plot_preds(x, y, f, model, title):
```

```
  plt.figure()
  plt.plot(x, y, '.', label='Data')
  plt.plot(x, f(x), label='Ground truth')
  plt.plot(x, model(x), label='Predictions')
  plt.title(title)
  plt.legend()

plot_preds(x, y, f, quad_model, 'Before training')
```



Tramite un processo di training voglio fare in modo che la curva approssimi quella iniziale. Per farlo definisco un modello quadratico con `loss function` pari allo scarto quadratico medio.

Dato che questo modello è progettato per prevedere valori continui, la mean squared error (MSE, errore quadratico medio) è una buona scelta come funzione di perdita. Date un vettore di previsioni $\hat{y}$ e un vettore di veri obiettivi y, l'MSE è definito come la media delle differenze quadrate tra i valori previsti e la verità del terreno.

$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$

dove m è il numero di campioni o esempi nei dati di addestramento, e $\hat{y}_i$ rappresenta la previsione del modello per il campione i e y_i rappresenta il valore obiettivo reale per lo stesso campione. L'MSE misura l'errore quadratico medio tra le previsioni del modello e i valori effettivi, con l'obiettivo di minimizzare questo errore durante l'addestramento del modello.

[41]:
```
def mse_loss(y_pred, y):
  return tf.reduce_mean(tf.square(y_pred - y))
```

Scriviamo un ciclo di addestramento di base per il modello. Il ciclo utilizzerà la funzione di perdita MSE e i suoi gradienti rispetto all'input per aggiornare iterativamente i parametri del modello. L'utilizzo di mini-batch per l'addestramento offre sia efficienza nella gestione della memoria che una convergenza più veloce. L'API `tf.data.Dataset` dispone di funzioni utili per il batching e il mescolamento.

Scrivo quindi un training loop, utilizzando mini batch di dimensione 32 per efficienza. TensorFlow offre una funzinoe ottimizzata per creare il dataset a partire da tensori.

Uso *shuffle* per riordinarli casualmente, *batch* per costruire i mini *batch*. Le epoche sono 100, c'è un learning rate pari a 0.01 Per ogni epoca scorriamo sul dataset (ottenendo un x batch e un Y batch, input e valori target associati). Per ogni mini batch costruisco un blocco gradient all'interno del quale passo in ingresso ad una funzione gli input e mi calcolo la *loss* in base a valori ottenuti e target.

Calcolo il gradiente della loss rispetto a tutte le variabili del modello. A questo punto utilizzo la *discesa del gradiente* per aggiornare i pesi del modello iterando su gradienti e varibili. Sottraggo quindi dalle varibili i learning rate per il gradiente. Posso memorizzare in una variabile la loss per ogni epoca per poi stamparla.

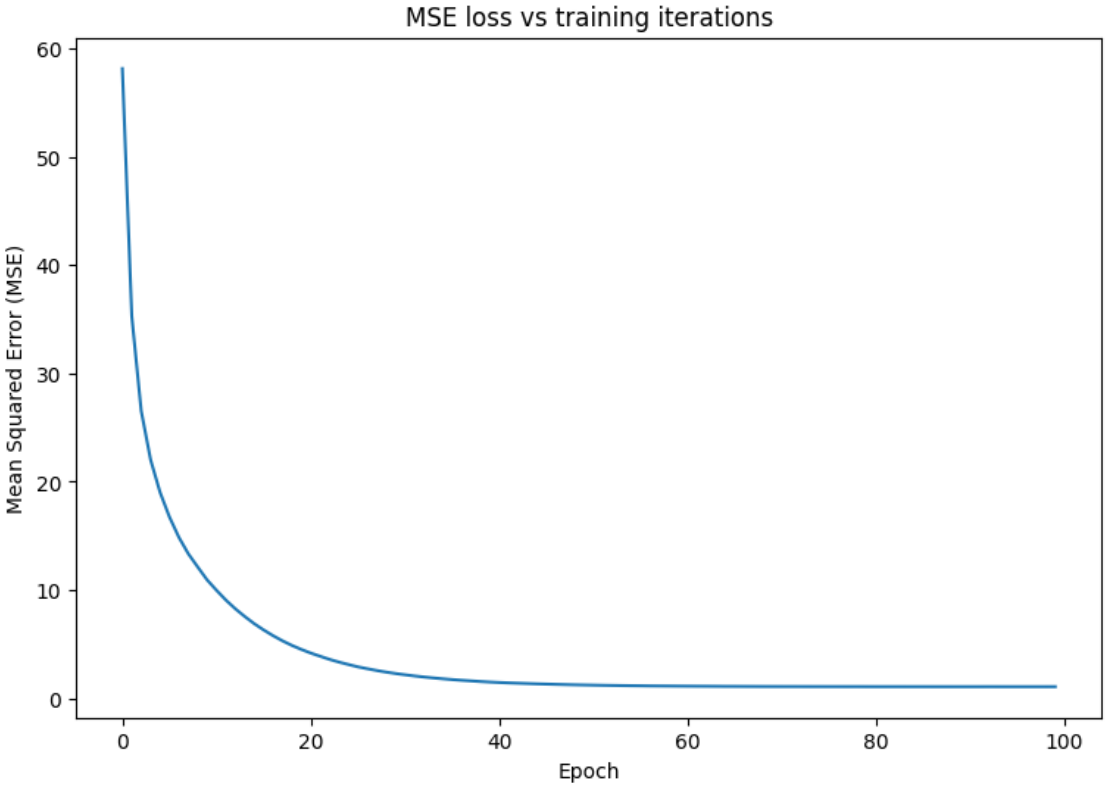
Le nuove predizioni funzionano bene rispetto al grafico atteso.

(Da qui in poi tutta la roba vista non la utilizzeremo, usiamo solo modelli e livelli).

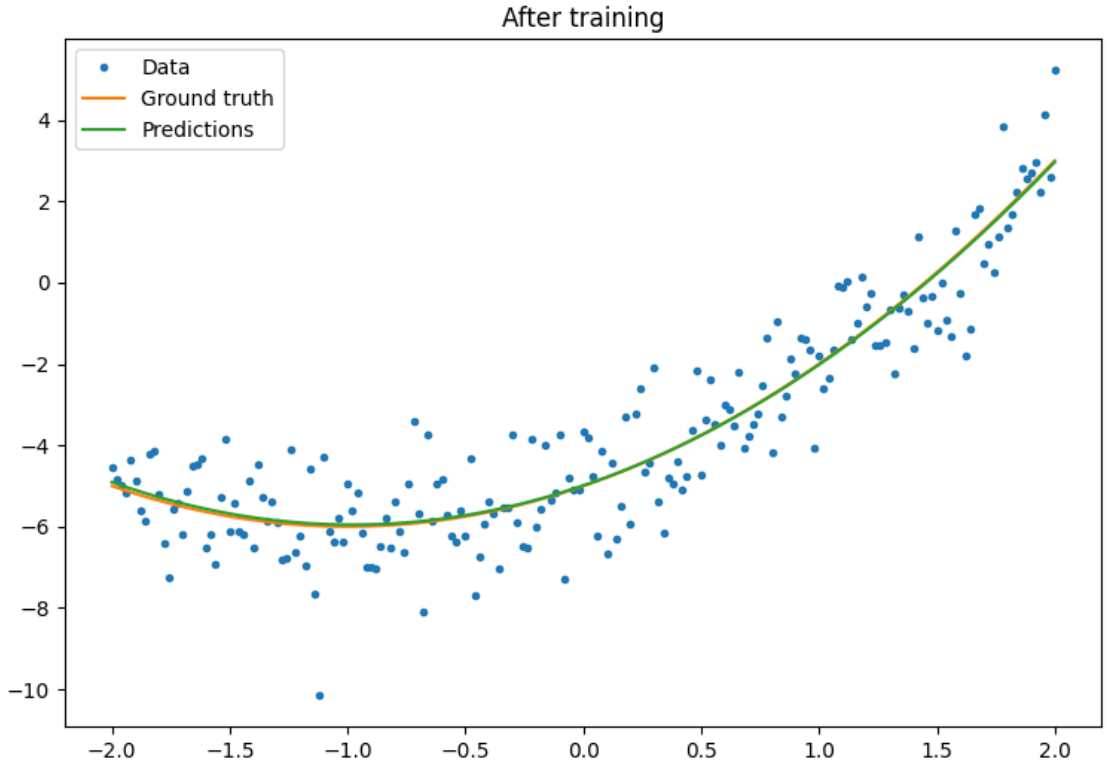
[42]:
```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices((x, y))
dataset = dataset.shuffle(buffer_size=x.shape[0]).batch(batch_size)
# Set training parameters
epochs = 100
learning_rate = 0.01
losses = []

# Format training loop
for epoch in range(epochs):
  for x_batch, y_batch in dataset:
    with tf.GradientTape() as tape:
      pred = quad_model(x_batch)
      batch_loss = mse_loss(pred, y_batch)
    # Update parameters with respect to the gradient calculations
    grads = tape.gradient(batch_loss, quad_model.variables)
    for g,v in zip(grads, quad_model.variables):
        v.assign_sub(learning_rate*g)
  # Keep track of model loss per epoch
  loss = mse_loss(quad_model(x), y)
  losses.append(loss)
  if epoch % 10 == 0:
    print(f'Mean squared error for step {epoch}: {loss.numpy():0.3f}')
```

```
# Plot model results
print("\n")
plt.plot(range(epochs), losses)
plt.xlabel("Epoch")
plt.ylabel("Mean Squared Error (MSE)")
plt.title('MSE loss vs training iterations');
```

```
Mean squared error for step 0: 58.119
Mean squared error for step 10: 9.983
Mean squared error for step 20: 4.197
Mean squared error for step 30: 2.196
Mean squared error for step 40: 1.484
Mean squared error for step 50: 1.241
Mean squared error for step 60: 1.147
Mean squared error for step 70: 1.113
Mean squared error for step 80: 1.102
Mean squared error for step 90: 1.099
```



[43]:
```
plot_preds(x, y, f, quad_model, 'After training')
```



È importante notare che, anche se l'implementazione di utilità comuni per l'addestramento è disponibile nel modulo `tf.keras`, è sempre consigliabile considerare l'uso di queste utilità prima di scrivere le proprie.

2.8 Keras

Offre astrazioni costruite su tensori e variabili: Il **livello** e il **modello**.

2.8.1 Livello

Il livello è definito come una trasformazione semplice input output, è una scatola nera che prende un input e produce un output (solitamente tensori). Ogni livello all'interno può avere pesi (stato interno), aggiornato durante il training. Può essere usato anche per rappresentare step di pre-processamento dei dati, ad esempio per normalizzare i dati in input (al suo interno non ha neuroni).

2.8.2 Modello

Il modello è un grafo diretto aciclico di livelli. Il modello sequenziale è particolamernete semplice, è uno stack di livelli in cui ognuno è connesso con il successivo.

Modello offre due metodi particolameten utili: - **fit()**: per addestarre il modello, nasconde tutto il loop di addestramento. - **predict()** per fare predizione.

Per costruire un modello sequenziale uso la classe `keras.Sequential` a cui passo un insieme di layers che verranno inseriti nel modello uno dopo l'altro. Il livello più semplice si chiama *Dense*,

il quale indica che tutti gli elementi del livello di input saranno interconnessi con opportuno peso con tutti gli elementi del livello considerato. Si può specificare la funzione di attivazione. Se non specificata sarà una combinazione lineare. Tiene conto anche del bias.

Si può definire un input, per poi calcolare y come `model(x)`.

[45]:
```
import warnings
warnings.filterwarnings('ignore')

import tensorflow as tf
from tensorflow import keras
from keras import layers
```

[46]:
```
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu"),
        layers.Dense(3, activation="relu"),
        layers.Dense(3, activation="sigmoid"),
    ]
)
# Call model on a test input
x = tf.ones((1,3))
y = model(x)
print(y)
```

```
tf.Tensor([[0.5 0.5 0.5]], shape=(1, 3), dtype=float32)
```

In modo equivalente si può definire un modello sequenziale aggiungendo con il metrodo add uno alla volta i vari livelli.

[47]:
```
model = keras.Sequential(name="mySequentialModel") # optional name
model.add(layers.Dense(2, activation="relu"))
model.add(layers.Dense(3, activation="relu"))
model.add(layers.Dense(3, activation="sigmoid"))
```

2.9 Particolarità

In generale, tutti i livelli in Keras devono conoscere la forma dei loro input al fine di poter creare i loro pesi. Quindi, quando si crea un livello in questo modo, inizialmente non ha pesi:

[48]:
```
layer = layers.Dense(3) #livello con tre elementi
layer.weights  # Empty
```

[48]:
```
[]
```

Non conoscendo l'input non so quanti pesi devo avere. Se istanzio il livello e basta ancora non sono inizializzati i pesi.

l livello crea i suoi pesi la prima volta che viene chiamato su un input, poiché la forma dei pesi dipende dalla forma degli input:

[49]:
```
# Call layer on a test input
x = tf.ones((1, 4))
y = layer(x)
layer.weights  # Now it has weights, of shape (4, 3) and (3,)
```

[49]:
```
[<tf.Variable 'dense_6/kernel:0' shape=(4, 3) dtype=float32, numpy=
 array([[-0.3191645 ,  0.38987577,  0.01320159],
        [-0.7827405 ,  0.5142399 , -0.05245537],
        [-0.33225507, -0.5579006 , -0.80228597],
        [-0.38001168,  0.22701526, -0.6426518 ]], dtype=float32)>,
 <tf.Variable 'dense_6/bias:0' shape=(3,) dtype=float32, numpy=array([0., 0.,
0.], dtype=float32)>]
```

In generale la cosa si applica ad un modello più complesso, i pesi verranno inizializzati solo una volta che passo i pesi al modello.

Naturalmente, ciò si applica anche ai modelli sequenziali. Quando si istanzia un modello sequenziale senza una forma di input, il modello non è "costruito": non ha pesi (e chiamare model.weights genera un errore che indica proprio questo). I pesi vengono creati quando il modello vede per la prima volta dei dati di input.

Una volta che un modello è "costruito", è possibile chiamare il suo metodo summary() per visualizzare il suo contenuto.

Tuttavia, può essere molto utile durante la costruzione incrementale di un modello sequenziale poter visualizzare il riepilogo del modello finora creato, compresa la forma corrente dell'output.

In questo caso, è possibile:

- Iniziare il modello passando un oggetto Input al modello, in modo che conosca la sua forma di input fin dall'inizio.
- Passare l'argomento input_shape al costruttore del livello.

[50]:
```
model = keras.Sequential()
model.add(keras.Input(shape=(4,)))
model.add(layers.Dense(2, activation="relu"))

# ALTERNATIVE:
#model = keras.Sequential()
#model.add(layers.Dense(2, activation="relu", input_shape=(4,)))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_7 (Dense)             (None, 2)                 10

=================================================================
Total params: 10 (40.00 Byte)
```

```
Trainable params: 10 (40.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

`model.summary()` mi restitusice i livelli che lo compongono, la dimensione dell'output (primo valore mi dice che lavoro su x istance di input), parametri che ha il modello (pesi+bias). Tutti i parametri sono addestrabili.

Il problema si può avere se ho costruito un modello e si vuole fare debugging. Se non è stato costruito non si può chiamare summary. Si può costruire uno pseudo-livello che dà le dimensioni dell'input. Così facendo si costruisce il modello e si può eseguire la summary (verrà restituito None come primo parametro dell'output shape).

Nota: come strategia di debug quando si definiscono modelli complessi, è utile chiamare frequentemente model.summary() tra le chiamate a .add() per verificare di non aver commesso errori.

2.9.1 Functional API

Per fare cose più sofisticate. Si utilizza il livello o il modello come se fosse una funzione applicata all'input. Costruisco un keras input, un oggetto dense che è un livello dense, definisco un tensore come risultato di applicazione del mio livello ad inputs e posso costruire un modello di output (sto costruendo pezzo per pezzo la rete neurale).

A questo punto posso costruire il modello passando inputs, outputs e nome modello.

Utilizzando questa API posso costruire un grafo più complesso della semplice sequenza di livelli.

Si può usare per fare layer sharing (costruire due modelli che condividono uno stesso livello).

L'API Functional di Keras è un modo per creare modelli più flessibili rispetto all'API keras.Sequential. L'API Functional può gestire modelli con topologie non lineari, livelli condivisi e persino più input o output.

L'idea principale è che un modello di deep learning è di solito un grafo aciclico diretto (DAG) di livelli. Quindi, l'API funzionale è un modo per costruire grafi di livelli.

Per costruire questo modello utilizzando l'API funzionale, inizia creando un nodo di input:

[51]:
```
# Just for demonstration purposes.
inputs = keras.Input(shape=(784,))
dense = layers.Dense(64, activation="relu")
x = dense(inputs)
x = layers.Dense(64, activation="relu")(x)
outputs = layers.Dense(10)(x)

model = keras.Model(inputs=inputs, outputs=outputs, name="myModel")
model.summary()
```

```
Model: "myModel"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 784)]             0
```

```
 dense_8 (Dense)             (None, 64)                50240

 dense_9 (Dense)             (None, 64)                4160

 dense_10 (Dense)            (None, 10)                650

=================================================================
Total params: 55050 (215.04 KB)
Trainable params: 55050 (215.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Training Supponiamo di aver costruito un modello. Per passare dati in input o gli passo array di numpy o tfdata.dataset.

La classe `Model` fornisce API integrate per l'addestramento e la convalida, che saranno sufficienti per la maggior parte delle esigenze. Per casi d'uso avanzati, è possibile implementare cicli di addestramento personalizzati.

Quando si passano dati ai cicli di addestramento integrati di un modello, è consigliabile utilizzare array NumPy (se i dati sono piccoli e rientrano in memoria) o oggetti tf.data.Dataset. Nei prossimi paragrafi, useremo il dataset MNIST come array NumPy, al fine di dimostrare come utilizzare ottimizzatori, funzioni di perdita e metriche

Esempio dataset MINIST (dataset cifre scritti a mano) Il database MNIST (Modified National Institute of Standards and Technology database) è un ampio database di cifre scritte a mano che viene comunemente utilizzato per addestrare vari sistemi di elaborazione delle immagini.

Consideriamo il modello seguente (qui lo costruiamo con l'API funzionale, ma potrebbe essere un modello Sequenziale o un modello sottoposto a subclassing): **Obiettivo**: riconoscere cifra nell'immagine. Costruisco un modello di rete feedforward che ha in input un vettore di 784 elementi (28*28). Prendo l'immagine e concateno i pixel in un unico vettore.

[52]:
```
inputs = keras.Input(shape=(784,), name="digits")
x = layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = layers.Dense(10, activation="softmax", name="predictions")(x)

model = keras.Model(inputs=inputs, outputs=outputs)
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 digits (InputLayer)         [(None, 784)]             0

 dense_1 (Dense)             (None, 64)                50240
```

```
 dense_2 (Dense)             (None, 64)                4160

 predictions (Dense)         (None, 10)                650

=================================================================
Total params: 55050 (215.04 KB)
Trainable params: 55050 (215.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- Il primo livello ha 64 unità e utilizza una **relu**.
- Il secondo livello uguale.
- Livello di uscita presenta 10 unità che utilizzano **softmax**.

In keras, per creare un livello di uscita, ho bisogno di un ulteriore livello nascosto che ha come numero di unità il numero di classi possibili e come funzione la softmax, che mi dà le probabilità.

Se stampo il summary si notano due livelli nascosti e il livello di uscita con 10 valori di uscita per un totale di 55000 parametri addestrabili.

A questo punto si divide il dataset caricato con funzione built-in (dataset che si usa sempre), lo divido in training e test set. Eseguo pre-processamento sui dati, andando a fare un *reshaping* per far diventare tutte le immagini (matrici di pixel) vettori (perché il modello si aspetta un vettore).

Divido per 255 perché per ogni pixel il dataset ha valori che stanno tra 0 biancoe 255 nero. Per lavorare con dati tra 0 e 1 noramlizzo.

Prendo una parte delle istanze del training set come validation.

[53]:
```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Preprocess the data (these are NumPy arrays)
# Note: reshape is used to convert the 28x28 image to a flat 784-element vector
x_train = x_train.reshape(60000, 784).astype("float32") / 255
x_test = x_test.reshape(10000, 784).astype("float32") / 255

y_train = y_train.astype("float32")
y_test = y_test.astype("float32")

# Reserve 10,000 samples for validation
x_val = x_train[-10000:]
y_val = y_train[-10000:]
x_train = x_train[:-10000]
y_train = y_train[:-10000]
display(y_train)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11490434/11490434 [==============================] - 2s 0us/step

array([5., 0., 4., …, 8., 4., 8.], dtype=float32)
```

Dobbiamo specificare la *configurazione del training*, ovvero 'optimizer,loss,metrics'.

[54]:
```
model.compile(
    optimizer=keras.optimizers.RMSprop(),  # Optimizer
    # Loss function to minimize
    loss=keras.losses.SparseCategoricalCrossentropy(),
    # List of metrics to monitor
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)

print("Fit model on training data")
history = model.fit(
    x_train,
    y_train,
    batch_size=64,
    epochs=2,
    # We pass some validation for
    # monitoring validation loss and metrics
    # at the end of each epoch
    validation_data=(x_val, y_val),
)
```

```
Fit model on training data
Epoch 1/2
782/782 [==============================] - 3s 3ms/step - loss: 0.3287 -
sparse_categorical_accuracy: 0.9062 - val_loss: 0.1810 -
val_sparse_categorical_accuracy: 0.9458
Epoch 2/2
782/782 [==============================] - 3s 3ms/step - loss: 0.1513 -
sparse_categorical_accuracy: 0.9549 - val_loss: 0.1362 -
val_sparse_categorical_accuracy: 0.9582
```

Devo utilizzare `compile` per specificare algoritmo da utilizzare per il training, la loss ed eventuali metriche da monitorare (es accurancy) durante il trainign (va fatto prima del fit).

A questo punto abbiamo chiamato la `fit`, passando dati di training (x e y, etichette), taglia del batch, epoche e validation set; in modo che tra le varie epoche ci si accorgesse di come stesse andando il modello.

Notiamo *782 iterazioni* all'interno dell'epoca, ma noi avevamo 7000 dati in input, questa disparità è motivata dal fatto che, ogni iterazione, si lavora su un minibatch. Abbiamo specificato *durata epoca*,*loss*, *accurancy* e altre informazioni.

Una volta addestrato il modello si possono valutare le prestazioni sul test set tramite evaluate.

Nell'esempio, abbiamo utilizzato l'argomento `validation_data` per passare una tupla di array NumPy (x_val, y_val) al modello al fine di valutare una perdita di convalida e le metriche di convalida alla fine di ciascuna epoca.

Ecco un'altra opzione: l'argomento `validation_split` consente di riservare automaticamente una parte dei dati di addestramento per la convalida. Il valore dell'argomento rappresenta la frazione

dei dati da riservare per la convalida, quindi dovrebbe essere impostato su un numero maggiore di 0 e minore di 1. Ad esempio, `validation_split=0.2` significa "utilizza il 20% dei dati per la convalida" e `validation_split=0.6` significa "utilizza il 60% dei dati per la convalida".

Il modo in cui viene calcolata la convalida è prendendo gli ultimi x% campioni degli array ricevuti dalla chiamata a `fit()`, prima di qualsiasi mescolamento.

Teniamo presente che puoi utilizzare validation_split solo quando addestri con dati NumPy.

[57]:
```
# Evaluate the model on the test data using `evaluate`
print("Evaluate on test data")
results = model.evaluate(x_test, y_test, batch_size=128)
print("test loss, test acc:", results)#les will be lost.


# Generate predictions (probabilities -- the output of the last layer)
# on new data using `predict`
print("Generate predictions for 3 samples")
predictions = model.predict(x_test[:3])
print("predictions shape:", predictions.shape)
```

```
Evaluate on test data
79/79 [==============================] - 0s 3ms/step - loss: 0.1353 -
sparse_categorical_accuracy: 0.9579
test loss, test acc: [0.13533227145671844, 0.9578999876976013]
Generate predictions for 3 samples
1/1 [==============================] - 0s 20ms/step
predictions shape: (3, 10)
```

2.9.2 Callbacks

Vengono specificate prima dell'addestramento e richiamate alla fine di ogni epoca, utili per *checkpoint* (e altre cose). Metodi implementati per fare quello che voglio durante il training. Consentono, ad esempio, di : - Salvare il modello a intervalli regolari o quando supera una determinata soglia di accuratezza. - Modificare il tasso di apprendimento del modello quando l'addestramento sembra essere in una fase di plateau. - Eseguire il fine-tuning dei livelli superiori quando l'addestramento sembra essere in una fase di plateau. - Inviare notifiche via email o messaggio istantaneo quando l'addestramento termina o supera una certa soglia di prestazioni. - Posso anche utilizzarle per far variare la `training_lenght` durante il training.

Callback predefiniti Ci sono molte callback predefinite già disponibili in Keras, come:

- `ModelCheckpoint`: Salva periodicamente il modello.
- `EarlyStopping`: Interrompe l'addestramento quando l'addestramento non migliora più le metriche di convalida.
- `TensorBoard`: Scrive periodicamente i log del modello che possono essere visualizzati in TensorBoard. Permette di visualizzare online i risultati durante il training.
- `CSVLogger`: Trasmette i dati di perdita e metriche su un file CSV. Creare un proprio callback.

Scrivere una callback propria È possibile creare un callback personalizzato estendendo la classe di base `keras.callbacks.Callback`. Un callback ha accesso al proprio modello associato attraverso la proprietà di classe `self.model`.

[ ]: