Deep Learning

Università Roma Tre
Dipartimento di Ingegneria
Anno Accademico 2022 - 2023

Kayers e moduli in Keras

Sommario

- Moduli e Keras
- Sequential
- Moduli custom
- Gestione dei parametri: lettura, condivisione, inizializzazione
- Inizializzazione lazy
- Layer custom
- I/O
- GPU e Keras

Deep networks e moduli

- Abbiamo visto come una MLP sia composta da uno o più layer, ognuno composto da uno o più nodi che costituiscono l'unità elementare di elaborazione.
- Alcuni risultati ci suggeriscono che questo sia un modello sufficientemente generale per simulare un dominio molto vasto funzioni. Ma risultati sperimentali hanno dimostrato che modelli intermedi, più grandi del singolo neurone, ma più piccoli dell'intero modello computazionale siano più adatti per costruire architetture deep.
  - Esempio: l'architettura *ResNet-152* (Residual NN) sviluppata nell'ambito della computer vision è una delle prime architetture con 100ia di layers. La rete è costituita da schemi di nodi e connessioni (**moduli**) che si ripetono. Particolari tecniche (*skip connections*) sono impiegate per risolvere il vanishing problem.
- Un modulo può essere un layer, più layers, o l'intero modello; e generalizza un elemento computazionale che può essere ripetuto, o riutilizzato in diverse architetture.

Moduli in Keras

- Un modulo è rappresentato da una classe Python che implementa la forward propagation, memorizza i parametri, e fornisca la backpropagation. Quest'ultimo aspetto può essere delegato alla tecnica autodiff, senza perciò definire manualmente i singoli gradienti.

- Ad esempio il seguente codice genera due layer: il primo con 256 nodi *fully connected* (o *denso*) ed uno di output con 10 nodi.

```
import tensorflow as tf

net = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dense(10),
])

X = tf.random.uniform((2, 20))
net(X).shape
```

- Il modello è costruito istanziando la classe *Sequential* e passandogli i singoli layer come parametri. Sia *Sequential* che *Dense* sono istanze di *keras.Model*.

Sequential in Keras

- Sequential crea una lista ordinata di layer.
- La procedura di forward propagation è definita implicitamente: l'output di un layer corrisponde all'input del secondo.
- Nell'esempio si invoca net(X), che corrisponde alla funzione net.call(X), per ottenere l'output dal modello appena creato.

Moduli custom

- Per creare nuovi moduli occorre tener presente come vengono impiegati durante l'esecuzione:

  1. I dati di input vengono mandati in input alla forward propagation
  2. La funzione di propagazione restituisce i valori in output
  3. Si calcolano i gradienti dell'output rispetto agli input per mezzo del metodo di backpropagation. Uno step solitamente gestito in automatico
  4. Memorizzare i parametri ottenuti necessari per la successiva forward propagation

Moduli custom

- Ad esempio, la rete precedente (un layer da 256 nodi seguito da un layer di 10 nodi) si codifica nel seguente modo:

```
class MLP(tf.keras.Model):
    def __init__(self):
        # Sempre necessario richiamare il costruttore della superclass
        super().__init__()
        self.hidden = tf.keras.layers.Dense(units=256, activation=tf.nn.relu)
        self.out = tf.keras.layers.Dense(units=10)

    # forward propagation
    def call(self, X):
        return self.out(self.hidden((X)))
```

- definendo il costruttore e la funzione che si occupa della forward propagation.

- Il metodo call permette di creare layer che richiedono particolari elaborazioni (es. controllare il flusso di esecuzione durante la forward propagation) che non corrispondono a quelle predefinite in Keras.

Moduli custom - esempio

- Durante l'elaborazione possiamo aver bisogno di accedere a costanti, cioè valori che non sono associati a parametri da stimare durante l'apprendimento, perciò non soggetti a back propagation.
- Nell'esempio, istanziamo i parametri in modo casuale, e rimarranno costanti durante il training. Restituiamo la somma dei valori in output.
- L'esempio è di scarsa utilità ma dimostra le potenzialità dei moduli custom.

```
class FixedHiddenMLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.flatten = tf.keras.layers.Flatten()
       self.rand_weight = tf.constant(tf.random.uniform((20, 20)))
        self.dense = tf.keras.layers.Dense(20, activation=tf.nn.relu)

    def call(self, inputs):
        X = self.flatten(inputs)
        # Usiamo i parametri costanti per generare l'output
        X = tf.nn.relu(tf.matmul(X, self.rand_weight) + 1)
        X = self.dense(X)
        # Control flow: simil l1 regularization
        while tf.reduce_sum(tf.math.abs(X)) > 1:
            X /= 2
        # reduce_sum() calcola la somma dei valori per una certa dimensione del tensore
        # senza secondo parametro la somma è operata su tutte le dimensioni del tensore
        return tf.reduce_sum(X)

net = FixedHiddenMLP()
net(X)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=0.88945085>
```

Moduli custom - esempio

- Nell'esempio definisco un altro modello (NestMLP) e successivamente un nuovo modello che include il primo come layer:

```
class NestMLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.net = tf.keras.Sequential()
        self.net.add(tf.keras.layers.Dense(64, activation=tf.nn.relu))
        self.net.add(tf.keras.layers.Dense(32, activation=tf.nn.relu))
        self.dense = tf.keras.layers.Dense(16, activation=tf.nn.relu)

    def call(self, inputs):
        return self.dense(self.net(inputs))

chimera = tf.keras.Sequential()
chimera.add(NestMLP())
chimera.add(tf.keras.layers.Dense(20))
chimera.add(FixedHiddenMLP())
chimera(X)
```

Esercizio

- Implementare un modulo che prende l'output di due moduli (es. *net1* e *net2*) e restituisce un output concatenato durante la forward propagation.

Gestione dei parametri: accesso

- Il loop di addestramento mira a trovare i parametri che minimizzano la funzione di loss. Le architetture più classiche hanno implementazioni che si occupano interamente della gestione dei parametri. In altri casi è necessario accedervi durante l'esecuzione (es. debugging, riuso dei parametri in parti diverse del modello).

- Vediamo come accedere ai parametri. Costruiamo un semplice modello:

```
import tensorflow as tf

net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4, activation=tf.nn.relu),
    tf.keras.layers.Dense(1),
])

X = tf.random.uniform((2, 4))
net(X).shape
```

- I layer nel modello sono memorizzati mediante liste. I parametri sono facilmente accedibili:

```
net.layers[2].weights       # secondo layer: 4 pesi e 1 bias
```

```
[<tf.Variable 'dense_1/kernel:0' shape=(4, 1) dtype=float32, numpy=
 array([[-0.6941955 ],
        [-0.9906301 ],
        [-0.13128954],
        [ 0.22367525]], dtype=float32)>,
 <tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32, numpy=array([0.], dtype=float32)>]
```

Gestione dei parametri: lettura

- In Keras i parametri sono salvati in particolari classi. Per ottenere il valore bisogna convertire le classi in tensori, ad esempio, per ottenere il bias dal secondo layer della rete:

```
type(net.layers[2].weights[1]), tf.convert_to_tensor(net.layers[2].weights[1])
```

```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.], dtype=float32)>)
```

- Mentre per ottenere tutti i parametri:

```
net.get_weights()
```

```
[array([[-0.20149094,  0.69364685, -0.12403131,  0.81778544],
        [ 0.3347332 ,  0.43645364,  0.18376476, -0.5020199 ],
        [-0.7681664 , -0.14477473, -0.6313741 ,  0.8246415 ],
        [-0.8074637 , -0.20050609,  0.4308104 ,  0.69257575]],
       dtype=float32),
 array([0., 0., 0., 0.], dtype=float32),
 array([[-0.6941955 ],
        [-0.9906301 ],
        [-0.13128954],
        [ 0.22367525]], dtype=float32),
 array([0.], dtype=float32)]
```

Condivisione dei parametri

- In alcune architetture è conveniente condividere i parametri in layer distinti, in modo che la modifica dei parametri di un layer si rifletta sull'altro.

```
shared = tf.keras.layers.Dense(4, activation=tf.nn.relu)
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    shared,
    shared,
    tf.keras.layers.Dense(1),
])
net(X)
```

- In questo caso, i gradienti del secondo e terzo layer sono sommati.

Inizializzazione dei parametri

- All'interno del modulo Python *keras.initializers* sono contenute le implementazioni di vari tipi di inizializzazione dei parametri. Tali approcci dipendono solitamente dall'input e dell'output e i valori dei bias sono impostati a zero.

- Per default, l'inizializziazione dei parametri è basata su una distribuzione uniforme (*glorot initializer*) nell'intervallo [-k,k], dove k = sqrt(6/(fin_in + fan_out)).

```
import tensorflow as tf

net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4, activation=tf.nn.relu),
    tf.keras.layers.Dense(1),
])

X = tf.random.uniform((2, 4))
net(X).shape
```

- Vedi: https://keras.io/api/layers/initializers/

Inizializzazione dei parametri

- Nell'esempio si impiega una inizializzazione basata su una distribuzione gaussiana con deviazione standard 0.01.

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4, activation=tf.nn.relu,
        kernel_initializer=tf.random_normal_initializer(mean=0, stddev=0.01),
        bias_initializer=tf.zeros_initializer()),
    tf.keras.layers.Dense(1)])

net(X)
net.weights[0], net.weights[1]
```

```
(<tf.Variable 'dense_2/kernel:0' shape=(4, 4) dtype=float32, numpy=
 array([[-0.00021173,  0.00316905, -0.00598176,  0.00144992],
        [-0.00882782,  0.01484077, -0.00652608, -0.00581241],
        [ 0.00398763, -0.01069997, -0.01145216, -0.00430671],
        [ 0.00342147, -0.01215916,  0.01345742,  0.01632656]],
       dtype=float32)>,
 <tf.Variable 'dense_2/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0., 0., 0.], dtype=float32)>)
```

- Invece per una inizializzazione con valori costanti:

```
    tf.keras.layers.Dense(
        4, activation=tf.nn.relu,
        kernel_initializer=tf.keras.initializers.Constant(1),
        bias_initializer=tf.zeros_initializer()),
```

- Nota: è possibile impiegare inizializzazioni distinte per ogni layer.

Inizializzazione dei parametri - custom

- Per una inizializzazione custom dei parametri bisogna creare una classe a partire dalla classe Initializer, e definire la funzione __call__() che restituisce il tensore in base alle dimensioni passate come parametro, es:

```
class MyInit(tf.keras.initializers.Initializer):
    def __call__(self, shape, dtype=None):
        data=tf.random.uniform(shape, -10, 10, dtype=dtype)
        factor=(tf.abs(data) >= 5)
        factor=tf.cast(factor, tf.float32)
        return data * factor

net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4,
        activation=tf.nn.relu,
        kernel_initializer=MyInit()),
    tf.keras.layers.Dense(1),
])

net(X)
print(net.layers[1].weights[0])
```

```
<tf.Variable 'dense_8/kernel:0' shape=(4, 4) dtype=float32, numpy=
array([[-0.       , -6.526873 ,  8.615063 ,  5.7617836],
       [ 0.       ,  0.       ,  6.0559807, -0.       ],
       [-6.7486644,  8.665197 ,  0.       , -7.035637 ],
       [-0.       , -0.       , -7.608464 ,  0.       ]], dtype=float32)>
```

Inizializzazione dei parametri - custom (2)

- In alternativa si possono impostare i parametri direttamente:

```
net.layers[1].weights[0][:].assign(net.layers[1].weights[0] + 1)
net.layers[1].weights[0][0, 0].assign(42)
net.layers[1].weights[0] # stampa

<tf.Variable 'dense_8/kernel:0' shape=(4, 4) dtype=float32, numpy=
array([[42.       , -5.526873 ,  9.615063 ,  6.7617836],
       [ 1.       ,  1.       ,  7.0559807,  1.       ],
       [-5.7486644,  9.665197 ,  1.       , -6.035637 ],
       [ 1.       ,  1.       , -6.608464 ,  1.       ]], dtype=float32)>
```

- Nell'esempio aggiorno i pesi del primo layer (+1) e imposto uno specifico peso al valore 42.

Inizializzazione lazy

- Nel codice visto, il risultato di alcune istruzioni dipende da iperparametri quali la dimensione dei layer (es. inizializzazione dei parametri, inserire un layer senza indicarne il numero di nodi), sebbene tali iperparametri non sono esplicitamente indicati.
- Con la inizializzazione differita (o lazy) è possibile definire una architettura in modo più possibile parametrico, in modo da specificare solo gli iperparametri essenziali e derivare gli altri in modo automatico.
- In questo esempio manca la dimensione del layer di input, perciò Keras non può definire completamente gli iperparametri:

```
import tensorflow as tf

net = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dense(10),
])

[net.layers[i].get_weights() for i in range(len(net.layers))]

[[], []]
```

- ...

Inizializzazione lazy (2)

- Se proviamo a definire le dimensioni di un certo input, Keras può completare l'inizializzazione, ad esempio:

```
X = tf.random.uniform((2, 20))
net(X)
[w.shape for w in net.get_weights()]
```

```
[(20, 256), (256,), (256, 10), (10,)]
```

Layer custom

- Possiamo definire layer anche senza parametri da sottoporre ad addestramento. Nell'esempio implementiamo una sorta di normalizzazione sottraendo la media dai valori in input. Tale operazioni vanno inserite nella funzione call().

```
import tensorflow as tf
from d2l import tensorflow as d2l


class CenteredLayer(tf.keras.Model):
    def __init__(self):
        super().__init__()

    def call(self, inputs):
        return inputs - tf.reduce_mean(inputs)


layer = CenteredLayer()
layer(tf.constant([1.0, 2, 3, 4, 5]))
```

```
<tf.Tensor: shape=(5,), dtype=float32, numpy=array([-2., -1.,  0.,  1.,  2.], dtype=float32)>
```

- Impieghiamo il layer custom nel nostro modello, e verifichiamo che con dait random otteniamo un valore medio in output quasi 0:

```
net = tf.keras.Sequential([tf.keras.layers.Dense(128), CenteredLayer()])

Y = net(tf.random.uniform((4, 8)))
tf.reduce_mean(Y)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=9.313226e-10>
```

Layer custom con parametri

- Nell'esempio ricreiamo un layer fully connected con una classe custom. Nella funzione *__init__*() creiamo i parametri che non dipendono dalla dimensione dell'input, mentre in *build*() definiamo quelli che dipendono, con eventuale inizializzazione. La funzione *build*() viene invocata automaticamente prima di call().

- La funzione *add_weight*() automatizza la creazione dei parametri da sottoporre ad addestramento.

```
class MyDense(tf.keras.Model):
    def __init__(self, units):
        super().__init__()
        # il secondo parametro indica la dimensione dell'input
        self.units = units

    # il secondo parametro indica la dimensione dell'input
    def build(self, X_shape):
        self.weight = self.add_weight(name='weight',
            shape=[X_shape[-1], self.units],
            initializer=tf.random_normal_initializer())
        self.bias = self.add_weight(
            name='bias', shape=[self.units],
            initializer=tf.zeros_initializer())

    def call(self, X):
        linear = tf.matmul(X, self.weight) + self.bias
        return tf.nn.relu(linear)

# vedi anche https://www.tensorflow.org/guide/keras/custom_layers_and_models
```

Layer con parametri

- Nell'esempio ricreiamo un layer fully connected con una classe custom. Nella funzione *__init__*() creiamo i parametri che non dipendono dalla dimensione dell'input, mentre in *build*() definiamo quelli che dipendono, con eventuale inizializzazione. La funzione *build*() viene invocata automaticamente prima di call().

- La funzione *add_weight*() automatizza la creazione dei parametri da sottoporre ad addestramento.

```
class MyDense(tf.keras.Model):
    def __init__(self, units):
        super().__init__()
        # il secondo parametro indica la dimensione dell'input
        self.units = units

    # il secondo parametro indica la dimensione dell'input
    def build(self, X_shape):
        self.weight = self.add_weight(name='weight',
            shape=[X_shape[-1], self.units],
            initializer=tf.random_normal_initializer())
        self.bias = self.add_weight(
            name='bias', shape=[self.units],
            initializer=tf.zeros_initializer())

    def call(self, X):
        linear = tf.matmul(X, self.weight) + self.bias
        return tf.nn.relu(linear)

# vedi anche https://www.tensorflow.org/guide/keras/custom_layers_and_models
```

I/O su file - tensori

- Gli addestramenti di reti deep possono essere molto lunghe. È necessario salvare i risultati parziale e finali su file in modo da poterli recuperare facilmente.

- Ad esempio, per salvare e recuperare i tensori:

```
import numpy as np
import tensorflow as tf

# salvataggio
x = tf.range(4)
np.save('x-file.npy', x)

# recupero
x2 = np.load('x-file.npy', allow_pickle=True)

# salvataggio di più sensori
y = tf.zeros(4)
np.save('xy-files.npy', [x, y])
x2, y2 = np.load('xy-files.npy', allow_pickle=True)

# o salvare dizionari stringa-tensore
mydict = {'x': x, 'y': y}
np.save('mydict.npy', mydict)
mydict2 = np.load('mydict.npy', allow_pickle=True)
```

I/O su file - modelli

- Per i modelli, occorre distinguere architettura e parametri. Per la prima, ci si basa sul codice che si usa per crearla, perciò senza salvataggio su file. Mentre per i parametri si sfruttano le funzionalità di Keras.

- Ad esempio, definiamo una architettura, salviamo i parametri e ricostruiamo la rete con il recupero dei parametri:

```
class MLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.flatten = tf.keras.layers.Flatten()
        self.hidden = tf.keras.layers.Dense(units=256, activation=tf.nn.relu)
        self.out = tf.keras.layers.Dense(units=10)

    def call(self, inputs):
        x = self.flatten(inputs)
        x = self.hidden(x)
        return self.out(x)

net = MLP()
X = tf.random.uniform((2, 20))
Y = net(X)

net.save_weights('mlp.params')

...

clone = MLP()
clone.load_weights('mlp.params')
```

GPU e Keras

- Per default i tensori sono creati in memoria e le computazioni sono sulla CPU. Ma possiamo comunque controllare l'elaborazione:

```
import tensorflow as tf
from d2l import tensorflow as d2l

def cpu():
    return tf.device('/CPU:0')

def gpu(i=0):
    return tf.device(f'/GPU:{i}')

cpu(), gpu(), gpu(1)
```

```
(<tensorflow.python.eager.context._EagerDeviceContext at 0x7fafa2b271c0>,
 <tensorflow.python.eager.context._EagerDeviceContext at 0x7fafa257a100>,
 [<tensorflow.python.eager.context._EagerDeviceContext at 0x7fafa253ad00>,
  <tensorflow.python.eager.context._EagerDeviceContext at 0x7fafa253ab40>])
```

```
def num_gpus():
    return len(tf.config.experimental.list_physical_devices('GPU'))

def try_gpu(i=0):
    # restituisce gpu(i) se esiste, altrimenti cpu()
    if num_gpus() >= i + 1:
        return gpu(i)
    return cpu()

def try_all_gpus():
    # Numero di GPU disponibili, o CPU se le GPU non ci sono
    return [gpu(i) for i in range(num_gpus())]

try_gpu(), try_gpu(10), try_all_gpus()
```

GPU e Keras (2)

```
# su quale device è allocato il tensore
# Nota: è fondamentale avere tutti i parametri di una operazione sullo stesso device
x = tf.constant([1, 2, 3])
x.device
```

```
'/job:localhost/replica:0/task:0/device:GPU:0'
```

```
# alloca un tensore su una GPU
with try_gpu():
    X = tf.ones((2, 3))
```

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

```
# alloca un tensore sulla seconda GPU
with try_gpu(1):
    Y = tf.random.uniform((2, 3))
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.44844735, 0.7493162 , 0.5692874 ],
       [0.10097635, 0.81023645, 0.5274769 ]], dtype=float32)>
```

```
# per calcolare X + Y, dobbiamo averli sullo stesso device
# spostiamo X sulla stessa GPU di Y
with try_gpu(1):
    Z = X
print(X)
print(Z)
```

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

```
# ora possiamo calcolarlo
Y + Z
```

GPU e Keras (3)

- È possibile indicare a Keras di impiegare le GPU disponibili per l'elaborazione di un certo modello:

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    net = tf.keras.models.Sequential([
        tf.keras.layers.Dense(1)])

INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/device:GPU:0', '/
job:localhost/replica:0/task:0/device:GPU:1')

net(X)

<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[-1.1522729],
       [-1.1522729]], dtype=float32)>

# vediamo la conferma cheanche i parametri sono memorizzati nello stesso device
net.layers[0].weights[0].device, net.layers[0].weights[1].device

('/job:localhost/replica:0/task:0/device:GPU:0',
 '/job:localhost/replica:0/task:0/device:GPU:0')
```