Distilling a Neural Network Into a Soft Decision Tree

`https://arxiv.org/pdf/1711.09784.pdf[1]`

Studente: Domenico Scognamiglio

Matricola: 0120/000169

1 Introduzione

L'obiettivo che si pone questo paper e' comprendere le motivazioni che, in funzione delle caratteristiche di un generico input, spingono una deep neural network ad effettuare una data classificazione. L'approccio basato sull'analisi del comportamento dei neuroni costituenti i vari hidden layer puo' risultare estremamente complesso, soprattutto nel caso di reti molto profonde, ed e' quindi stato proposto un approccio alternativo basato sugli alberi binari di decisione. Infatti, a differenza di una classica *dnn*, e' molto semplice comprendere le motivazioni che spingono un albero di decisione ad effettuare una data classificazione, in quanto e' sufficiente l'analisi di una breve sequenza di decisioni, ognuna dipendente esclusivamente dai dati di input. L'idea proposta consiste nel costruire un albero binario di decisione, nel quale ogni nodo interno e' un neurone collegato direttamente all'input e ogni foglia e' un "bigotto" che, indipendentemente dall'input, produce sempre la stessa distribuzione di probabilita' delle possibili classi di output. In seguito all' addestramento di questo albero binario e' possibile non solo classificare nuovi dati in modo decisamente piu' veloce rispetto ad una *dnn* ma, analizzando i neuroni appartenenti al percorso radice-foglia che porta ad una data classificazione, e' possibile comprendere le caratteristiche dell'input che l'hanno determinata. Il drawback in questo caso e' rappresentato dall'accuratezza che risultera' essere inferiore rispetto ad una *dnn*. Anche se cio' non risulta essere l'obiettivo principale di questo lavoro, viene comunque utilizzata una tecnica (*distillation*[2]) che consente di trasferire in parte le abilita' di generalizzazione di una *dnn* all'albero in modo da migliorarne l'accuratezza.
Segue una descrizione sommaria del software progettato in Python basato sul noto framework Tensorflow (versione 1.5).

2 Analisi del codice sorgente

2.1 Placeholders creation

Il primo passo consiste nella creazione dei placeholders, tensori a dimensione variabile i cui valori verranno iniettati al momento dell'esecuzione:

```
1 def create_placeholders(n_x,n_y):
2     X = tf.placeholder(tf.float64,[None,n_x],name="X")
3     Y = tf.placeholder(tf.float64, [None, n_y], name="Y")
4     V_prec = tf.placeholder(tf.float64, [None], name="V_prec")
5     return X,Y,V_prec
```

`Args`:

- `n_x`: numero di features dell'input.
- `n_y`: numero di classi.

`Returns`:

- `X`: tensore 2-d corrispondente all'input, costituito da un numero variabile di righe (in quanto a runtime a seconda del tipo di strategia utilizzata potrebbe variare) e da un numero di colonne pari al numero di feature dell'input.
- `Y`: tensore 2-d corrispondente alle label costituito anch'esso da un numero variabile di righe e da un numero di colonne pari al numero di classi (one-hot encoding).
- `V_prec`: tensore 1-d utilizzato per mantenere le medie mobili durante la fase di regolarizzazione.

$$\mathbf{X} = \underbrace{\left.\begin{pmatrix} {x_0}^{[0]} & {x_1}^{[0]} & \cdots & {x_{n_x}}^{[0]} \\ {x_0}^{[1]} & {x_1}^{[1]} & \cdots & {x_{n_x}}^{[1]} \\ \vdots & & \ddots & \\ {x_0}^{[m]} & {x_1}^{[m]} & \cdots & {x_{n_x}}^{[m]} \end{pmatrix}\right\}}_{\text{input features}} \text{samples} \qquad \mathbf{Y} = \underbrace{\left.\begin{pmatrix} 0 & 0 & 1 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}\right\}}_{\text{classes}} \text{samples}$$

m:minibatch size.

2.2 Parameters initialization

Vengono successivamente creati i tensori corrispondenti ai pesi/bias dei neuroni (nodi interni dell'albero) e ai parametri relativi alle singole foglie:

```
def initialize_parameters(n_x):
    W = tf.get_variable("W", [n_x,inner_numb], initializer=tf.contrib.layers.
        xavier_initializer(),dtype=tf.float64)
    b = tf.get_variable("b", [1, inner_numb], initializer=tf.zeros_initializer(),dtype=tf.
        float64)
    phi = tf.get_variable("phi", [leafs_numb,k], initializer=tf.contrib.layers.
        xavier_initializer(),dtype=tf.float64)
    parameters = {"W": W,
                  "b": b,
                  "phi":phi}
    return parameters
```

`Args`:

- `n_x`: numero di features dell'input.

`Returns`:

- `parameters`: dizionario con i riferimenti ai tensori `W` (pesi), `b` (bias) e `phi` (parametri foglie).

$$\mathbf{W} = \underbrace{\left.\begin{pmatrix} {w_0}^{[0]} & {w_1}^{[0]} & \cdots & {w_{I_n}}^{[0]} \\ {w_0}^{[1]} & {w_1}^{[1]} & \cdots & {w_{I_n}}^{[1]} \\ \vdots & & \ddots & \\ {w_0}^{[n_x]} & {w_1}^{[n_x]} & \cdots & {w_{I_n}}^{[n_x]} \end{pmatrix}\right\}}_{\text{inner nodes}} \text{input features} \qquad \mathbf{B} = \underbrace{\begin{pmatrix} b_0 & b_1 & \cdots & b_{I_n} \end{pmatrix}}_{\text{inner nodes}}$$

I_n :numero di nodi interni dell'albero.

$$\boldsymbol{\phi} = \underbrace{\begin{pmatrix} {\phi_0}^{[0]} & {\phi_0}^{[1]} & \cdots & {\phi_0}^{[k]} \\ {\phi_1}^{[0]} & {\phi_1}^{[1]} & \cdots & {\phi_1}^{[k]} \\ \vdots & & \ddots & \\ {\phi_{L_n}}^{[0]} & {\phi_{L_n}}^{[1]} & \cdots & {\phi_{L_n}}^{[k]} \end{pmatrix}}_{\text{classes}} \Bigg\} \text{ leafs}$$

L_n : numero di foglie dell'albero, k : numero di classi.

2.3 Forward propagation

In seguito vengono calcolate le attivazioni dei neuroni corrispondenti ai vari input:

```python
def forward_propagation(X, parameters):
    W = parameters['W']
    b = parameters['b']
    Z= tf.add(tf.matmul(X,W),b)
    A=tf.sigmoid(Z) #activations of inner nodes
    return A
```

`Args`:

- `X`: tensore 2-d relativo ai dati di input.
- `parameters`: dizionario dei parametri.

`Returns`:

- `A`: tensore 2-d relativo alle attivazioni dei neuroni.

$$\boldsymbol{A} = \underbrace{\begin{pmatrix} {a_0}^{[0]} & {a_1}^{[0]} & \cdots & {a_{I_n}}^{[0]} \\ {a_0}^{[1]} & {a_1}^{[1]} & \cdots & {a_{I_n}}^{[1]} \\ \vdots & & \ddots & \\ {a_0}^{[m]} & {a_1}^{[m]} & \cdots & {a_{I_n}}^{[m]} \end{pmatrix}}_{\text{inner nodes}} \Bigg\} \text{ samples}$$

Dove il generico elemento ${a_i}^{[j]}$ rappresenta l'attivazione del neurone i rispetto all'input $x^{[j]}$.

$${a_i}^{[j]} = \sigma\left(x^{[j]} \cdot w_i + b_i\right)$$

L'albero utilizzato, come descritto dal paper, e' un soft decision tree ossia un albero nel quale le decisioni vengono prese su base probabilistica. In particolare, dato un input j, il valore ${a_i}^{[j]}$ rappresenta la probabilita' di discesa sul figlio destro del nodo i, mentre il valore $1 - {a_i}^{[j]}$ quella di discesa sul figlio sinistro dello stesso nodo i. Ogni foglia invece apprende una distribuzione di probabilita' statica sulle possibili k classi di output espressa mediante i parametri contenuti nel tensore $\boldsymbol{\phi}$ illustrato in precedenza. Data una foglia ℓ, la probabilita' che tale foglia assegnera' ad una classe k sara' pari a:

$${Q_k}^{\ell} = \frac{exp\left({\phi_\ell}^{[k]}\right)}{\sum_{k'} exp\left({\phi_\ell}^{[k']}\right)}$$

Ossia ad una softmax calcolata sulla riga ℓ della matrice $\boldsymbol{\phi}$.

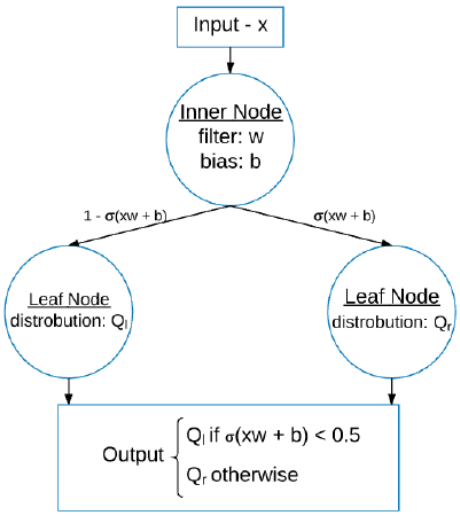

Figura 1: Processo di decisione

Per classificare un generico input quindi si parte dalla radice e, note le attivazioni di tutti i neuroni per tale input, si scende sempre lungo il ramo piu' probabile tra i due possibili. Infine, giunti ad una foglia, viene effettuata la classificazione scegliendo la classe piu' probabile sulla base dei parametri appresi dalla foglia stessa. Dato un input x, la loss function e' definita nel modo seguente:

$$L(x) = -\sum_{\ell \in LeafNodes} P^{\ell}(x) \sum_{k} T_k \log Q_{\ell}^{[k]}$$

Il termine $P^{\ell}(x)$ indica la probabilita' di giungere attraverso il processo di decisione alla foglia ℓ a partire dall'input x, e puo' essere ottenuta dalla produttoria delle probabilita' delle decisioni incluse nel percorso radice-foglia. Il secondo termine e' semplicemente la cross-entropy tra la softmax calcolata sui parametri della foglia ℓ e la label (espressa in forma one-hot) del campione x. Attraverso la minimizzazione della loss function si cerca quindi di fare in modo che ogni foglia si specializzi per una singola classe e che ogni input venga direzionato verso una foglia corretta con elevata probabilita'.

2.4 Leafs cross-entropy matrix

Mediante la seguente function `compute_leafs_cross_entropy_matrix()` calcolo la cross-entropy tra le foglie e le label:

```
def compute_leafs_cross_entropy_matrix(phi,Y):
    m=tf.shape(Y)[0]
    ta = tf.TensorArray(dtype=tf.float64, size=leafs_numb)
    init_state = (0, ta)
    condition = lambda i, _: i < leafs_numb
    # tf.tile because each leafs is a "bigot" that always produce the same distribution
    body = lambda i, ta: (i + 1, ta.write(i,tf.nn.softmax_cross_entropy_with_logits(logits=
        tf.tile(tf.reshape(phi[i,:],[1,k]),[m,1]),labels=Y)))
    n, ta_final = tf.while_loop(condition, body, init_state)
    leafs_ce_matrix = ta_final.stack()
    return leafs_ce_matrix
```

`Args`:

- `phi`: tensore 2-d contenente i parametri delle foglie.

- `Y`: tensore 2-d contenente le label

`Returns`:

- `leafs_ce_matrix`: tensore 2-d contenente i valori della cross-entropy calcolata tra ciascuna foglia e ciascun campione con label in `Y`.

Per ogni foglia viene prima eseguito il tiling dei parametri, replicandoli m volte sulle righe in modo da costruire un tensore 2-d che abbia lo stesso shape del tensore `Y` e successivamente, mediante la function `tf.nn.softmax_cross_entropy_with_logits()`, viene applicata una softmax a tali parametri e viene calcolata la cross-entropy con le label in `Y`. Il tensore 1-d ottenuto viene infine inserito in un `TensorArray`. Effettuando lo stack dei tensori ottenuti per le varie foglie otteniamo un tensore 2-d seguente:

$$\boldsymbol{LEAFS\_CE\_MATRIX} = \underbrace{\left.\begin{pmatrix} CE\left(\ell_0, y^{[0]}\right) & CE\left(\ell_0, y^{[1]}\right) & \cdots & CE\left(\ell_0, y^{[m]}\right) \\ CE\left(\ell_1, y^{[0]}\right) & CE\left(\ell_1, y^{[1]}\right) & \cdots & CE\left(\ell_1, y^{[m]}\right) \\ \vdots & & \ddots & \\ CE\left(\ell_{L_n}, y^{[0]}\right) & CE\left(\ell_{L_n}, y^{[1]}\right) & \cdots & CE\left(\ell_{L_n}, y^{[m]}\right) \end{pmatrix}\right\}}_{\text{samples}} \text{ leafs}$$

2.5 Leafs probability matrix

Ai fini del calcolo della cost-function, e' necessario anche calcolare, come citato in precedenza, le probabilita' di foglie e nodi interni (queste ai fini della regolarizzazione) in funzione dei vari input. La function che segue consente di calcolare la probabilita' di giungere ad un nodo dell'albero in relazione agli m input con attivazioni in `A`:

```
def node_probability(index_node,A):
    p=tf.ones([tf.shape(A)[0]],dtype=tf.float64)
    init_state=(index_node,p)
    condition = lambda index_node,_:index_node-1 >= 0
    def body(index_node,p):
        father_index=tf.floor_div(index_node-1,2)
        return(father_index,p*tf.cond(tf.equal((index_node - 1) % 2, 0),
                                      lambda: 1.0 - A[:,father_index],
                                      lambda: A[:,father_index]))
    n,p_final=tf.while_loop(condition,body,init_state)
    return p_final
```

`Args`:

- `index_node`: indice del nodo dell'albero di cui si desidera calcolare le probabilita'.
- `A`: tensore 2-d contenente le attivazioni dei nodi interni.

`Returns`:

- `p_final`: tensore 1-d contenente le probabilita' del nodo avente indice pari a `index_node`, rispetto agli `m` input con attivazioni in `A`.

Per quanto riguarda gli indici dei nodi e' stata utilizzata la seguente convenzione. I nodi interni dell'albero hanno indici che vanno da 0 a $L_n - 2$, mentre le foglie hanno indici da 0 a $L_n - 1$. In tal modo e' possibile accedere direttamente ai tensori 2-d relativi a foglie e attivazioni descritti in precedenza. L'indice del padre di un nodo i, ricorrendo alla rappresentazione mediante array di un albero binario, sara' pari a $\lfloor (i-1)/2 \rfloor$

e attraverso l'analisi del resto e' possibile anche stabilire se tale nodo e' figlio destro o sinistro del padre. Nel ciclo while della function `node_probability()` vengono moltiplicati tra loro i tensori colonna relativi ai padri del nodo `index_node`, contenuti nel tensore 2-d delle attivazioni, ottenendo in tal modo il tensore 1-d contenente le probabilita' desiderate per il nodo in questione. Invocando tale function una volta per ogni foglia viene costruito il tensore 2-d contenente le probabilita' delle foglie per ciascuno degli m input:

```
def compute_leafs_prob_matrix(A):
    ta = tf.TensorArray(dtype=tf.float64, size=leafs_numb)
    init_state = (0, ta)
    condition = lambda i, _: i < leafs_numb
    body = lambda i,ta: (i+1,ta.write(i,node_probability(leafs_numb-1+i,A)))
    n,ta_final=tf.while_loop(condition,body,init_state)
    leafs_prob_matrix=ta_final.stack()
    return leafs_prob_matrix
```

Args:

- `A`: tensore 2-d contenente le attivazioni dei nodi interni.

Returns:

- `leafs_prob_matrix`: tensore 2-d contenente le probabilita' delle foglie per ciascuno degli `m` input con attivazioni in `A`.

$$\boldsymbol{LEAFS\_PROB\_MATRIX} = \underbrace{\left.\begin{pmatrix} P\left(\ell_0, x^{[0]}\right) & P\left(\ell_0, x^{[1]}\right) & \cdots & P\left(\ell_0, x^{[m]}\right) \\ P\left(\ell_1, x^{[0]}\right) & P\left(\ell_1, x^{[1]}\right) & \cdots & P\left(\ell_1, x^{[m]}\right) \\ \vdots & & \ddots & \\ P\left(\ell_{L_n}, x^{[0]}\right) & P\left(\ell_{L_n}, x^{[1]}\right) & \cdots & P\left(\ell_{L_n}, x^{[m]}\right) \end{pmatrix}\right\}}_{\text{samples}} \text{leafs}$$

2.6 Cost-function

A questo punto e' possibile procedere con il calcolo della cost-function illustrata in precedenza:

```
def compute_cost(A,phi,Y,V_prec):
    leafs_ce_matrix=compute_leafs_cross_entropy_matrix(phi,Y)
    leafs_prob_matrix=compute_leafs_prob_matrix(A)
    inner_prob_matrix=compute_inner_prob_matrix(A)
    #cost without regularization
    cost_wr=tf.reduce_mean(tf.reduce_sum(tf.multiply(leafs_ce_matrix,leafs_prob_matrix),0))
    reg,V_next=compute_regularization(A,inner_prob_matrix,V_prec)
    cost=cost_wr+reg
    return cost,V_next
```

Args:

- `A`: tensore 2-d contenente le attivazioni dei nodi interni.
- `phi`: tensore 2-d contenente i parametri delle foglie.
- `Y`: tensore 2-d contenente le label.
- `V_prec`: tensore 1-d utilizzato per mantenere le medie mobili durante la fase di regolarizzazione.

Returns:

- `cost`: cost-function calcolata sugli `m` input con attivazioni in `A`.
- `V_next`: tensore 1-d contenente le medie mobili da utilizzare al prossimo passo in fase di regolarizzazione.

Tralasciando gli aspetti legati alla regolarizzazione che verranno discussi in seguito, la function `compute_cost()` calcola il costo innanzitutto effettuando il prodotto element-wise tra i 2 tensori 2-d relativi alle cross-entropy delle foglie e alle probabilita' di queste rispetto agli `m` campioni di input. Viene in seguito effettuata la somma per righe del risultato, ottenendo in tal modo un tensore 1-d contenente le loss per ciascuno degli `m` input e infine viene calcolato il costo calcolando la media di quest'ultimo.

2.7 Regularization

L'obiettivo della regolarizzazione, come illustrato nel paper, consiste nell'introduzione di una penalita' che incoraggia ogni nodo ad utilizzare in modo equiprobabile entrambi i suoi sottoalberi destro e sinistro. Si vuole quindi fare in modo che l'output di un nodo, che come visto in precedenza determina la probabilita' di discesa a destra, sia simile il piu' possibile a 0.5. Questo deve essere valido soprattutto per i nodi che si trovano ai livelli superiori dell'albero. Considerando invece i nodi che si trovano ad una profondita' maggiore cio' perde di significato in quanto ci si aspetta che tali nodi si specializzino e quindi siano in grado di prendere decisioni nette. La penalita' attribuita sara' quindi direttamente proporzionale a 2^{-d} dove d indica la profondita' del nodo nell'albero. Dato quindi un batch di input, la penalita' applicata ad un generico nodo i viene calcolata attraverso la cross-entropy tra la distribuzione desiderata (0.5,0.5) e quella attuale $(a_i, 1-a_i)$, dove il termine a_i viene calcolato come segue:

$$\alpha_i = \frac{\sum_x P^i(x)\, p_i(x)}{\sum_x P^i(x)}$$

dove $P^i(x)$ indica la probabilita' di giungere al nodo i dato l'input x e $p_i(x)$ e' l'attivazione del nodo i dato l'input x. Il termine α_i esprime quindi la media pesata delle attivazioni del nodo i dove i pesi sono costituiti dalle probabilita' di giungere a tale nodo. La penalita' totale si ottiene nel modo seguente:

$$C = -\lambda \sum_{i \in InnerNodes} 0.5 \log(\alpha_i) + 0.5 \log(1-\alpha_i)$$

La probabilita' di giungere ad un nodo, decresce esponenzialmente con l'aumentare della profondita' del nodo stesso e di conseguenza il termine α_i tende a diventare meno accurato per i nodi piu' profondi dell'albero. Per risolvere cio', in [1] e' stato proposto un approccio basato sull'utilizzo di medie mobili esponenziali con finestre temporali direttamente proporzionali alle profondita' dei nodi. Ricordiamo innanzitutto la formula generica relativa ad una media mobile esponenziale:

$$V_t = \beta V_{t-1} + (1-\beta)\,\theta_t$$

Dove $V_0 = 0$, $\theta_1, \theta_2, ..., \theta_m$ sono i valori che si presentano nel tempo (nel nostro caso gli a_i tra i vari minibatch) e $\beta \in [0, 1)$ rappresenta l'ampiezza della finestra e regola l'importanza che hanno i dati passati nel determinare l'attuale valore della media. La funzione scelta per mappare l'ampiezza della finestra in funzione della profondita' di un nodo e' la seguente:

$$\beta_i = 1 - e^{-depth_i}$$

```python
def compute_regularization(A,inner_prob_matrix,V_prec):
    #tensorarray for regularization terms of inner nodes
    ta = tf.TensorArray(dtype=tf.float64, size=inner_numb)
    # tensorarray for maintaining moving averages
    ema = tf.TensorArray(dtype=tf.float64, size=inner_numb)
    init_state = (0,ta,ema)
    condition = lambda i,_,_2: i < inner_numb
    def body(i,ta,ema):
        depth = tf.floor(tf.log(tf.cast(i,tf.float64) + 1)/tf.log(tf.constant(2,dtype=tf.
            float64)))
        # decay is exponentially proportional to the depth of the inner node and is used for
             the time windows of the exponential decaying moving average as described in the
             paper.
        decay = 1. - tf.exp(-depth)
        a_i = tf.truediv(tf.tensordot(inner_prob_matrix[i, :], A[:, i], axes=1), tf.
            reduce_sum(inner_prob_matrix[i, :]))
        w_i=decay*V_prec[i]+(1.-decay)*a_i
        r_i = tf.reshape( -lmbd * (2 ** (-depth)) * (0.5 * tf.log(w_i) + 0.5 * tf.log(1.0 -
            w_i)), [1])
        return (i+1,ta.write(i,r_i),ema.write(i,w_i))
    n,ta_final,ema_final=tf.while_loop(condition,body,init_state)
    regularization=tf.reduce_sum(ta_final.stack())
    V_next=tf.reshape(ema_final.stack(),[inner_numb])
    return regularization,V_next
```

`Args`:

- `A`: tensore 2-d contenente le attivazioni dei nodi interni.
- `inner_prob_matrix`: tensore 2-d contenente le probabilita' dei nodi interni dell'albero rispetto agli `m` input con attivazioni in `A`.
- `V_prec`: tensore 1-d utilizzato per mantenere le medie mobili durante la fase di regolarizzazione.

`Returns`:

- `regularization`: termine di regolarizzazione da sommare al costo.
- `V_next`: tensore 1-d contenente le medie mobili da utilizzare al prossimo passo in fase di regolarizzazione.

Il tensore 2-d `inner_prob_matrix` viene ottenuto allo stesso modo illustrato in precedenza per le foglie.

$$\boldsymbol{INNER\_PROB\_MATRIX} = \underbrace{\left.\begin{pmatrix} P\left(i_0, x^{[0]}\right) & P\left(i_0, x^{[1]}\right) & \cdots & P\left(i_0, x^{[m]}\right) \\ P\left(i_1, x^{[0]}\right) & P\left(i_1, x^{[1]}\right) & \cdots & P\left(i_1, x^{[m]}\right) \\ \vdots & & \ddots & \\ P\left(i_{I_n}, x^{[0]}\right) & P\left(i_{I_n}, x^{[1]}\right) & \cdots & P\left(i_{I_n}, x^{[m]}\right) \end{pmatrix}\right\}}_{\text{samples}} \text{inner nodes}$$

Nel ciclo while innanzitutto viene determinato il termine α_i per ogni nodo interno. Il numeratore puo' essere ottenuto attraverso il prodotto scalare tra i due tensori 1-d relativi alle attivazioni e alle probabilita' del nodo i mentre per il denominatore basta effettuare la somma dei termini appartenenti al tensore 1-d delle probabilita del nodo stesso. Successivamente in `w_i` viene aggiornato il valore della media mobile esponenziale, utilizzando il valore della media al passo precedente (`V_prec[i]`) ricevuto in input. Infine viene calcolato il valore r_i che rappresenta la cross-entropy tra $(\alpha_i, 1-\alpha_i)$ e $(0.5, 0.5)$. Vengono quindi

costruiti due `TensorArray` che conterranno i valori `r_i` e `w_i` per ogni nodo interno dell'albero. Il termine di regolarizzazione finale viene ottenuto effettuando la somma dei vari `r_i`.

2.8 Soft decision tree training

Il training e' stato eseguito sul dataset `MNIST` su un soft binary decision tree avente profondita' pari a 8. Per la minimizzazione della cost-function e' stato utilizzato l'algoritmo di `Adam` su mini-batch di size 64.

```
#Hyperparameters
tree_levels=9
leafs_numb = 2**(tree_levels-1)
inner_numb = leafs_numb-1
k = 10
lmbd=0.1
alpha=0.01
num_epochs=41
minibatch_size=64

#Load MNIST dataset
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
train_data = mnist.train.images #shape(55.000,784)
train_labels = np.asarray(mnist.train.labels, dtype=np.int32) #shape(55.000,10)
cv_data = mnist.validation.images #shape(5.000,784)
cv_labels = np.asarray(mnist.validation.labels, dtype=np.int32) #shape(5.000,10)
eval_data = mnist.test.images #shape(10.000,784)
eval_labels = np.asarray(mnist.test.labels, dtype=np.int32) #shape(10.000,10)

(m,n_x)= train_data.shape
n_y= train_labels.shape[1]
costs=[]
X,Y,V_prec=create_placeholders(n_x,n_y)
parameters=initialize_parameters(n_x)
A=forward_propagation(X,parameters)
cost,V_next=compute_cost(A,parameters['phi'],Y,V_prec)
optimizer=tf.train.AdamOptimizer(learning_rate=alpha).minimize(cost,var_list=parameters)
#For tensorboard summary
summary_cost = tf.placeholder(tf.float64, shape=(), name="cost")
summary_training_accuracy = tf.placeholder(tf.float64, shape=(), name="tr_acc")
summary_test_accuracy = tf.placeholder(tf.float64, shape=(), name="test_acc")
tf.summary.scalar("COST",summary_cost)
tf.summary.scalar("TRAINING ACCURACY",summary_training_accuracy)
tf.summary.scalar("TEST ACCURACY",summary_test_accuracy)
add_weights_to_summary(parameters)
merged_summary_op = tf.summary.merge_all()
#---------------

init = tf.global_variables_initializer()
saver = tf.train.Saver(max_to_keep=100)
with tf.Session() as session:
    writer = tf.summary.FileWriter("logs/depth9", session.graph)
    # V_next_value is used for the exponential moving average of a_i across various
        minibatches in training phase
    V_next_value=np.zeros(inner_numb)
    session.run(init)
    for epoch in range(num_epochs):
        epoch_cost=0
        num_minibatches=int(m/minibatch_size)
        for minibatch in range(num_minibatches):
```

```
            minibatch_x, minibatch_y = mnist.train.next_batch(minibatch_size, shuffle=True)
            vn,_,minibatch_cost=session.run([V_next,optimizer,cost],feed_dict={X:minibatch_x
                ,Y:minibatch_y,V_prec:V_next_value})
            epoch_cost+=minibatch_cost/num_minibatches
            V_next_value=vn
        print("Cost after epoch %i: %f" % (epoch, epoch_cost))
        if epoch % 5 == 0:
            save_path = saver.save(session,"tmp/depth9/model.ckpt",global_step=epoch,
                write_meta_graph=False)
        par=session.run(parameters)
        training_accuracy=compute_accuracy(par,train_data,train_labels)
        test_accuracy=compute_accuracy(par, eval_data, eval_labels)
        print("Accuracy on training set:",training_accuracy)
        print("Accuracy on test set:", test_accuracy)
        summary = session.run(merged_summary_op,feed_dict={summary_cost:epoch_cost,
                                                           summary_training_accuracy:
                                                               training_accuracy,
                                                           summary_test_accuracy:
                                                               test_accuracy})
        writer.add_summary(summary, epoch)
    phi=session.run(parameters['phi'])
    print_leafs_distributions(phi)
```

Il miglior risultato ottenuto da vari test effettuati con iperparametri valorizzati come mostrato nel codice sorgente, e' stata un'accuratezza sul test set pari al 95.47% su un training di 23 epoche.

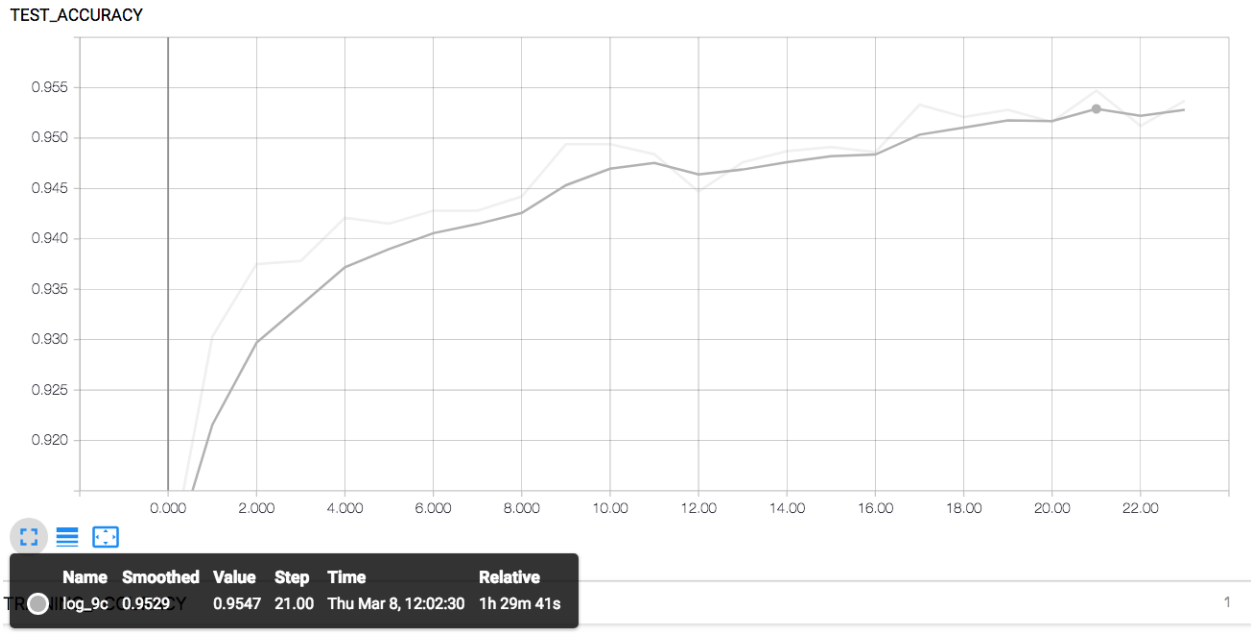

Figura 2: Test set accuracy

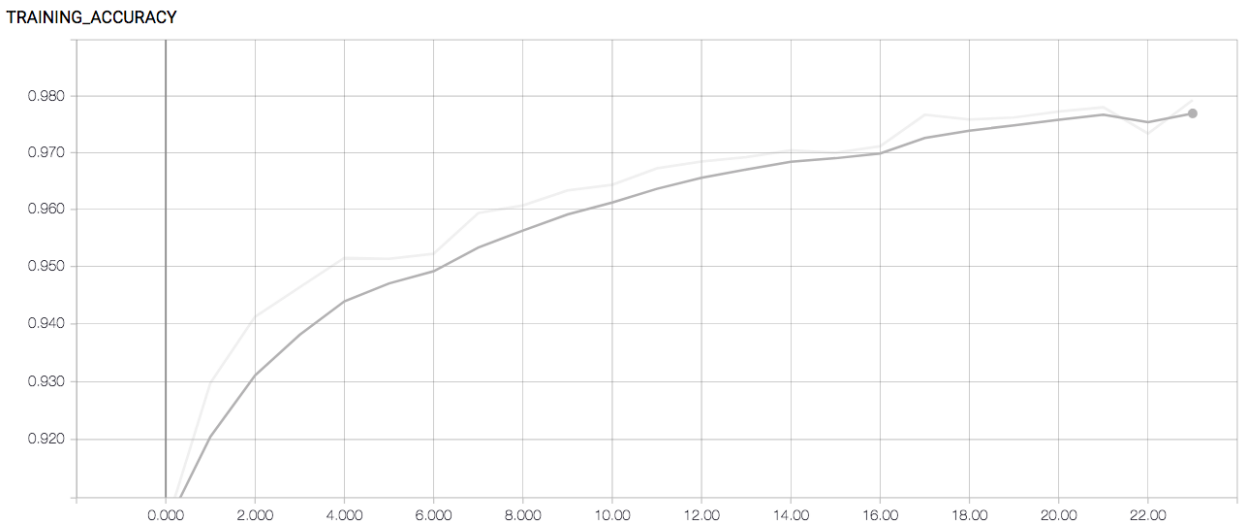

Figura 3: Training set accuracy

2.9 Distilling

Come descritto in [1] , si puo' aumentare l'accuratezza del modello addestrandolo su una combinazione di hard-label (quelle usate anche in precedenza per il training) e soft-label, quest'ultime prodotte da una rete neurale piu' grande caratterizzata da uno strato di output con un'alta temperatura nella softmax. La tecnica consiste nella minimizzazione di una cost-function composta da due funzioni obiettivo. La prima e' la classica cross-entropy con le hard-label, mentre la seconda e' la cross-entropy con le soft-label, calcolata utilizzando nel modello distillato la stessa temperatura della softmax utilizzata per il training della rete piu' grande. In tal modo ci si aspetta di trasferire parte della conoscenza e delle capacita' di generalizzazione dal modello grande a quello distillato piu' piccolo. La rete convolutiva utilizzata per produrre le soft label del dataset `MNIST` e' la seguente:

```
def MnistNetworkTeacher(input, keep_prob_conv, keep_prob_hidden, scope='Mnist', reuse=False):
    input=tf.cast(input,tf.float32)
    with tf.variable_scope(scope, reuse=reuse) as sc:
        with slim.arg_scope([slim.conv2d], kernel_size=[3, 3], stride=[1, 1],
                            biases_initializer=tf.constant_initializer(0.0), activation_fn=
                                tf.nn.relu):
            net = slim.conv2d(input, 32, scope='conv1')
            net = slim.max_pool2d(net, [2, 2], 2, scope='pool1')
            net = tf.nn.dropout(net, keep_prob_conv)

            net = slim.conv2d(net, 64, scope='conv2')
            net = slim.max_pool2d(net, [2, 2], 2, scope='pool2')
            net = tf.nn.dropout(net, keep_prob_conv)

            net = slim.conv2d(net, 128, scope='conv3')
            net = slim.max_pool2d(net, [2, 2], 2, scope='pool3')
            net = tf.nn.dropout(net, keep_prob_conv)

            net = slim.flatten(net)
        with slim.arg_scope([slim.fully_connected], biases_initializer=tf.
            constant_initializer(0.0),
                            activation_fn=tf.nn.relu):
            net = slim.fully_connected(net, 625, scope='fc1')
```

```
            net = tf.nn.dropout(net, keep_prob_hidden)
            net = slim.fully_connected(net, 10, activation_fn=None, scope='fc2')

            net = tf.nn.softmax(net / temperature)
            return net
```

Una volta eseguito il training di tale rete, vengono utilizzate come descritto le soft label prodotte per il training del soft decision tree. La funzione per il calcolo della cross-entropy e' stata quindi modificata nel modo che segue:

```
def compute_leafs_cross_entropy_matrix(phi,Y,soft_Y):
    m=tf.shape(Y)[0]
    ta = tf.TensorArray(dtype=tf.float64, size=leafs_numb)
    init_state = (0, ta)
    condition = lambda i, _: i < leafs_numb
    body = lambda i, ta: (i + 1, ta.write(i,0.5*tf.nn.softmax_cross_entropy_with_logits(
        logits=tf.tile(tf.reshape(phi[i,:],[1,k]),[m,1]),labels=Y)+1*(-tf.reduce_sum(tf.log(
        tf.nn.softmax(tf.truediv(tf.tile(tf.reshape(phi[i,:],[1,k]),[m,1]),temperature)))*tf
        .cast(soft_Y,tf.float64),axis=1))))
    n, ta_final = tf.while_loop(condition, body, init_state)
    leafs_ce_matrix = ta_final.stack()
    return leafs_ce_matrix
```

In alcuni test effettuati non sono tuttavia stati riscontrati ulteriori miglioramenti dell'accuratezza sul test-set rispetto a quanto ottenuto in precedenza con il training esclusivo sulle hard-label. Bisogna tuttavia considerare che tale tecnica introduce nuovi iperparametri (il cui tuning e' fondamentale) di cui non sono stati specificati i valori nel caso di esempio del `MNIST` citato in [1], dove grazie al distilling si e' ottenuta un'accuratezza del 96.76% sul test-set. Tali iperparametri sono rappresentati in primo luogo dalla temperatura della softmax utilizzata nel training della rete piu' grande, che determina quanto le label prodotte da tale rete siano soft, e in secondo luogo dai pesi attribuiti alle due funzioni obiettivo nella funzione costo finale (nel codice mostrato in precedenza sono stati attribuiti come pesi 0.5 e 1). Bisognerebbe effettuare test piu' approfonditi esplorando ulteriormente lo spazio degli iperparametri citati in modo da giungere possibilmente a risultati ancora migliori per quanto concerne l'accuratezza del modello.

Riferimenti bibliografici

[1] N. Frosst and G. E. Hinton, "Distilling a neural network into a soft decision tree," *CoRR*, vol. abs/1711.09784, 2017.

[2] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.