

Lezione 3 08/10/2024

Un'abilità importante di un modello è quella di **generalizzare**, in un dataset. È per questo che si fa training su un subset e poi si vede tramite il test set se il modello è in grado di generalizzare.

Per generalizzare, si usa la funzione di regolarizzazione.

La **loss function** misura la differenza tra il valore predetto e il valore reale del training set.

Discesa del gradiente stocastica

L'algoritmo più usato per ottimizzare è la **discesa del gradiente stocastica**. È un'estensione dell'algoritmo della discesa del gradiente.

Un problema ricorrente è che abbiamo bisogno di data set molto grande per avere una buona generalizzazione, ma dataset grandi sono costosi.

Caratteristica	Discesa del Gradiente (Batch)	Discesa del Gradiente Stocastica (SGD)
Calcolo del gradiente	Su tutto il dataset	Su un singolo esempio
Aggiornamento dei parametri	Una volta per iterazione (dopo il calcolo completo)	Dopo ogni esempio
Velocità di ogni iterazione	Lenta (richiede l'intero dataset)	Veloce (richiede un solo esempio)
Convergenza	Più stabile e regolare	Più irregolare, ma potenzialmente più veloce
Efficienza computazionale	Elevata su dataset grandi	Più efficiente su dataset grandi

La cost function può essere scritta in questo modo:

The cost function used by a Machine Learning algorithm often decomposes as a sum over n training examples of some per-example loss function.

For example, the negative conditional log-likelihood of the training data can be written as:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{data}} L(\mathbf{x}, y, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

where L is the per-example loss $L(\mathbf{x}, y, \boldsymbol{\theta}) = -\log p(y|\mathbf{x}; \boldsymbol{\theta})$.

Per calcolare il gradiente stocastico, dobbiamo calcolare l'approssimazione del gradiente.

For this additive cost functions, gradient descent requires the computation of:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

Il costo computazionale di questa operazione è $O(n)$ e quindi molto costoso se n è grande.

L'idea è la seguente: al posto di calcolare il gradiente di tutta la loss function e ottenere il gradiente esatto, posso iniziare ad approssimare il gradiente usando set di esempi piccoli, quindi senza calcolarlo tutto insieme ma calcolandolo e migliorandolo pian piano.

Il fatto che il gradiente è un "expectation" si può sfruttare in questo modo.

"Online" significa che calcolo l'approssimazione aggiornata del gradiente ad ogni sample.

Algorithm 2.1 Online stochastic gradient descent training.

Input:

- Function $f(\mathbf{x}; \Theta)$ parameterized with parameters Θ .
 - Training set of inputs $\mathbf{x}_1, \dots, \mathbf{x}_n$ and desired outputs y_1, \dots, y_n .
 - Loss function L .
-

```
1: while stopping criteria not met do
2:   Sample a training example  $\mathbf{x}_i, y_i$  ← It starts by sampling a training example
3:   Compute the loss  $L(f(\mathbf{x}_i; \Theta), y_i)$ 
4:    $\hat{\mathbf{g}} \leftarrow$  gradients of  $L(f(\mathbf{x}_i; \Theta), y_i)$  w.r.t  $\Theta$ 
5:    $\Theta \leftarrow \Theta - \eta_t \hat{\mathbf{g}}$ 
6: return  $\Theta$ 
```

Poi calcola la loss (3), e calcola il gradiente (4)

It computes the gradient w.r.t Θ

Input and predictions are assumed
to be fixed and the loss is treated
as a function of Θ

Poi (5)

Parameters Θ are updated in the
opposite direction of the gradient

scaled by a learning rate η_t

L'errore è basato in un singolo esempio di training, e quindi è un'estimazione
grossolana dell'intero dataset che vogliamo minimizzare, e quindi può trovare
gradienti inaccurati.

Stochastic Gradient Descent (SGD)

For this additive cost functions, gradient descent requires the computation of:

$$\nabla_{\theta} J(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L(x^{(i)}, y^{(i)}, \theta)$$

The computational cost of this operation is $O(n)$ – very expensive when n is large

But ...the gradient is an expectation and the expectation can be computed by using a small set of samples !!!

On each step of the algorithm, we can sample a **minibatch** of examples

$$\mathcal{B} = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$$

drawn from the training set.

m is typically chosen to be relatively small, ranging from 1 to a few hundred examples

Per risolvere questo problema usiamo il **minibatch**.

The estimate of the gradient is formed as

$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\theta} L(x^{(i)}, y^{(i)}, \theta)$$

Using the examples in $\mathcal{B} = \{x^{(1)}, x^{(2)}, \dots, x^{(m')}\}$

The stochastic gradient descent algorithm then follows the estimated gradient downhill

$$\theta \leftarrow \theta - \eta g$$

Where η is the learning rate

Questo è l'algoritmo del **minibatch stochastic gradient descent**:

Algorithm 2.2 Minibatch stochastic gradient descent training.

Input:

- Function $f(\mathbf{x}; \Theta)$ parameterized with parameters Θ .
- Training set of inputs $\mathbf{x}_1, \dots, \mathbf{x}_n$ and desired outputs y_1, \dots, y_n .
- Loss function L .

```
1: while stopping criteria not met do
2:   Sample a minibatch of  $m$  examples  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$ 
3:    $\hat{\mathbf{g}} \leftarrow 0$ 
4:   for  $i = 1$  to  $m$  do
5:     Compute the loss  $L(f(\mathbf{x}_i; \Theta), y_i)$ 
6:      $\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} + \text{gradients of } \frac{1}{m}L(f(\mathbf{x}_i; \Theta), y_i) \text{ w.r.t } \Theta$ 
7:    $\Theta \leftarrow \Theta - \eta_t \hat{\mathbf{g}}$ 
8: return  $\Theta$ 
```

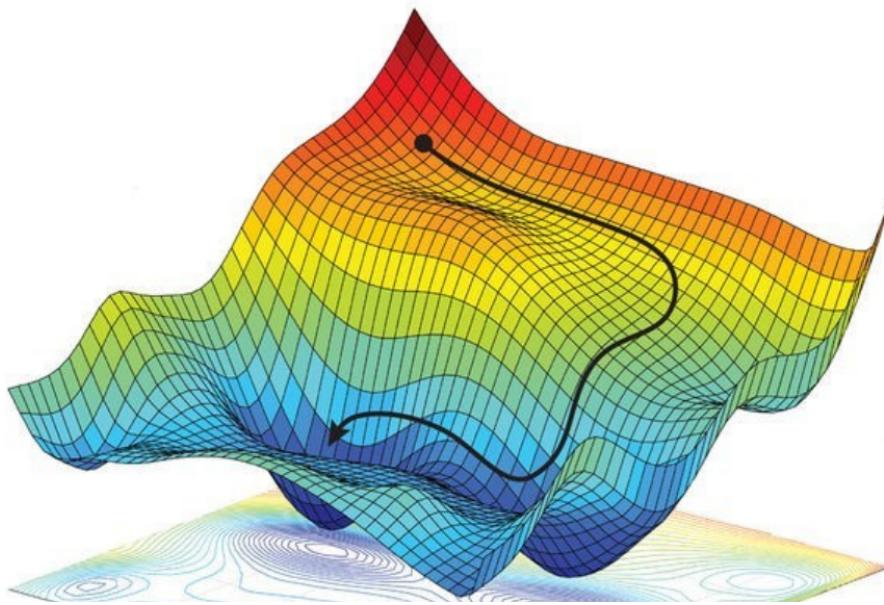
Estimate of the training loss based
on the minibatch

Quindi:

- Valori alti di m generano stime del gradiente migliori
- Valori bassi di m permettono di avere più updates e quindi una convergenza più veloce
- Permette un training efficiente perchè per valori piccoli di m alcune architetture come le GPU permettono un'implementazione in parallelo del gradiente

L'idea quindi è quella di diminuire lo step size più tardi nel training.

Loss functions



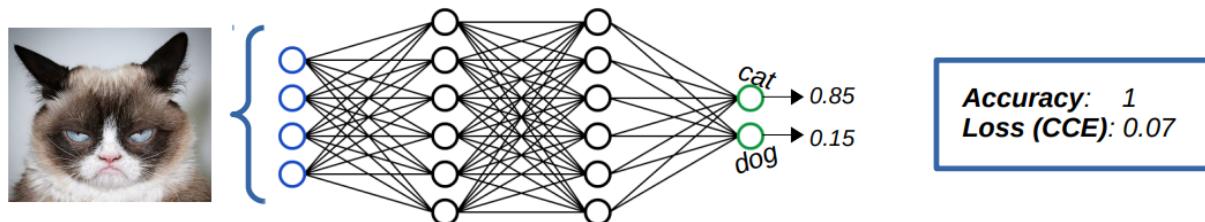
Per capire quanto è buono un neural network calcoliamo gli output e li compariamo con i target.

La loss function e la cost function sono spesso considerate come la stessa cosa, ma sono diverse:

- **la loss function** misura la performance della rete su un singolo datapoint.
- **la cost function** è la media delle loss sull'intero dataset.

Il nostro obiettivo è quello di minimizzare la cost function, usando batches.

La cost function ritorna un numero scalare, generalmente ≥ 0 (se =0 allora la performance è perfetta). Più basso è questo numero, meglio è.



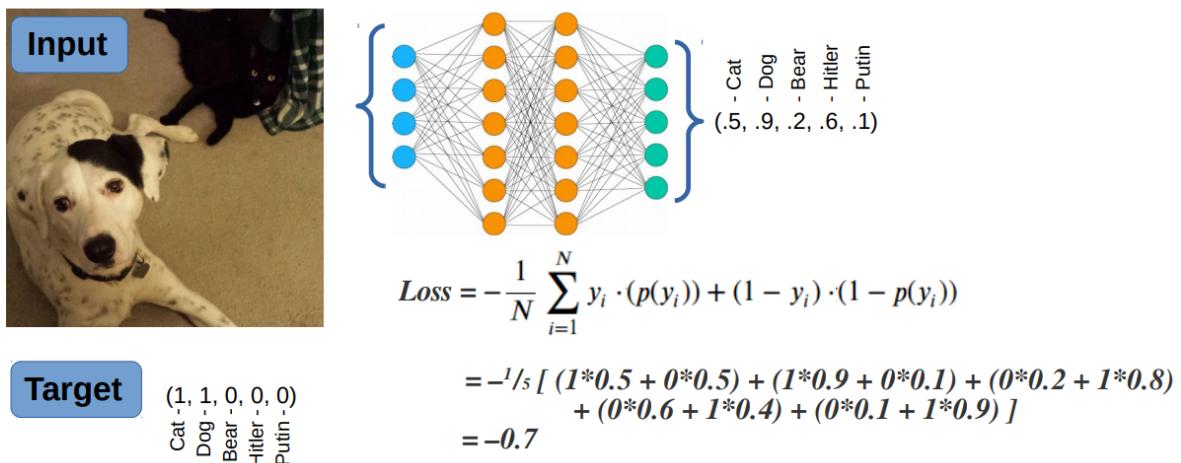
Ci sono diversi tipi di loss function:

- Classification
 - Maximum likelihood

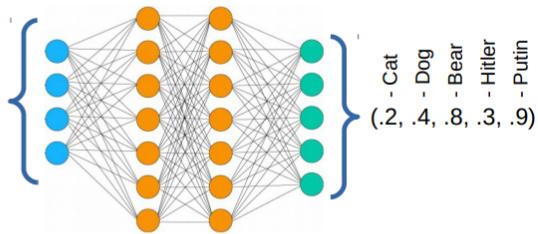
- Binary cross-entropy (aka log loss)
- Categorical cross-entropy
- Regression (i.e. function approximation)
 - Mean Squared Error
 - Mean Absolute Error
 - Huber Loss

Classification loss functions

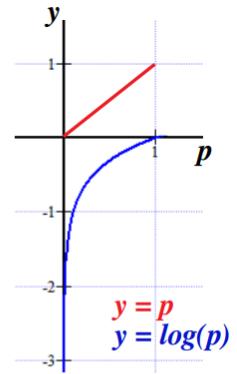
Maximum likelihood



Binary cross-entropy (aka Log loss)



Torch.nn.BCELoss



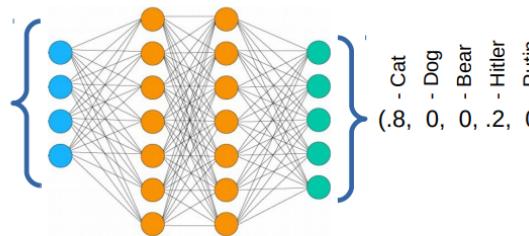
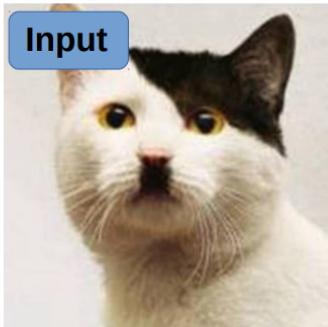
$$Loss = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i))$$

Target

Cat - (0, 0, 1, 0, 1)
Dog -
Bear -
Hitler -
Putin -

$$= -1/5 [(1 * \log(0.8)) + (1 * \log(0.6)) + (1 * \log(0.8)) + (1 * \log(0.7)) + (1 * \log(0.9))] \\ = 0.28$$

Categorical cross-entropy



Torch.nn.NLLLoss

Torch.nn.CrossEntropyLoss

This one does the softmax for you
 $= \log \left(\frac{\exp(x_i)}{\sum_j \exp(x_j)} \right)$

Must be a probability distribution

Target

Cat - (1, 0, 0, 0, 0)
Dog -
Bear -
Hitler -
Putin -

$$Loss = -\sum_{i=1}^N y_i \cdot \log(p(y_i))$$

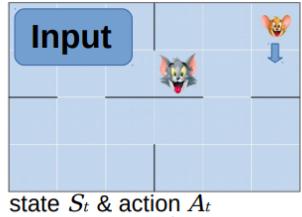
$$= - [1 * \log(0.8)] \\ = 0.22$$

This loss function can also be used when you have a single binary (e.g. yes/no) output

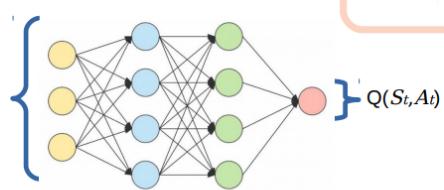
Regression loss functions

Nei problemi di regressione avremo un valore continuo come risultato, non un vettore (come nei problemi di classificazione).

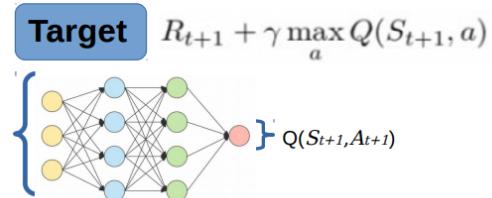
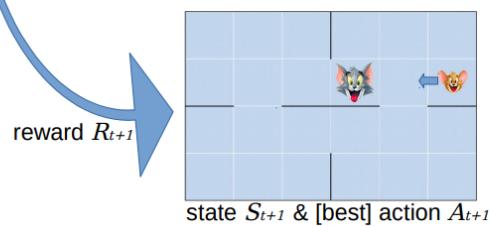
Mean squared error



Torch.nn.MSELoss



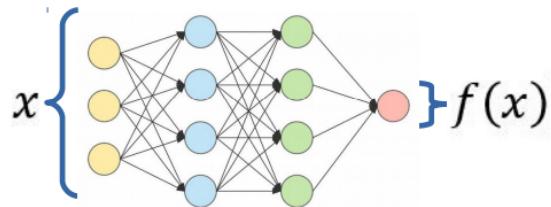
$$\begin{aligned} \text{Loss} &= (y - f(x))^2 \\ &= ((R_{t+1} + \gamma \max_a Q(S_{t+1}, a)) - Q(S_t, A_t))^2 \end{aligned}$$



Absolute error

Torch.nn.L1Loss

Input x



Target y

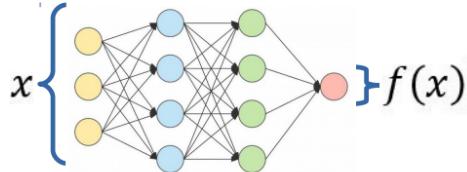
$$\text{Loss} = |y - f(x)|$$

Regression loss

Huber loss

Torch.nn.SmoothL1Loss

Input x



Target y

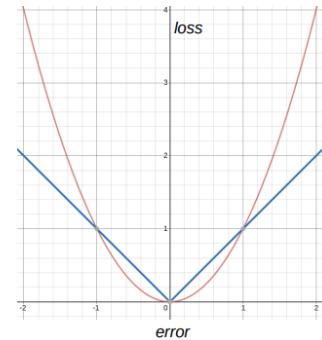
$$\text{Loss} = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

Huber loss combines the best features of MSE and AE:

- like MSE for small errors.
- like AE otherwise.

This avoids over-shooting:

- MSE has a **very steep gradient when the error is large** (i.e. for outliers) because of its quadratic form.
- AE has a **steep gradient when the error is small** because it doesn't have a 'flat' bottom.



Output functions

Abbiamo bisogno di loss functions che funzionano con il gradiente, ovvero per esempio quelle che non hanno salti.

Tutte le funzioni che si possono usare come output possono essere usate anche per i neuroni nascosti. Il ruolo dell'output è quello di fornire la trasformazione finale per le features, per completare la task del modello.

Output Units

- Linear for estimating the mean of Gaussian Distributions
 $\rightarrow 0 \in \mathbb{R}$ (`FALSE` è `TRUE`)
- Sigmoid units for Bernoulli output distributions
 \rightarrow CATEGORIA
- Softmax Units for Multinoulli Output distributions
- Gaussian Mixtures (Mixture density networks) for multimodal regression

Linear units for Gaussian Output distributions

Un tipo semplice di output è basato sul modello lineare:

$$\hat{\mathbf{y}} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

I layer di output lineari sono spesso usati per produrre la media di una distribuzione gaussiana condizionale:

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I})$$

Maximizing the log-likelihood is then equivalent to minimizing the mean squared error

But we have to learn the covariance matrix of the Gaussian; maximum likelihood makes it straightforward to learn that covariance as long as the covariance is a positive definite matrix for all inputs.

è difficile soddisfare vincoli come questi, con un output layer lineare, quindi tipicamente vengono usate altre unità di output per parametrizzare la covarianza.

Un'altra unità di output che è usata di solito è quella della sigmoide. In questo caso...

Sigmoid Units for Bernoulli Output Distributions

Molte task richiedono di predire il valore di una variabile binaria y

A Bernoulli distribution is defined by just a single number $p(y = 1|\mathbf{x})$ that must be in the range [0,1]

If we use a linear unit, we can threshold its value to obtain a valid probability:

$$p(y = 1|\mathbf{x}) = \max \{0, \min\{1, \omega^T \mathbf{h} + b\}\}$$

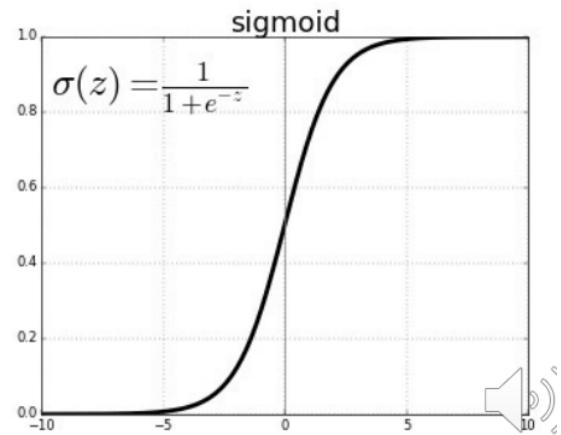
Però la funzione avrebbe gradiente 0 al di fuori dell'intervallo [0,1], quindi non va bene per il training. Per il training servono funzioni che non raggiungono un punto di saturazione velocemente.

Una scelta migliore è una funzione sigmoide, per assicurarsi un gradiente forte quando il modello ha la risposta sbagliata.

$$\hat{y} = \sigma(\omega^T h + b)$$

First compute $z = \omega^T h + b$ and then convert it into a probability

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



Sigmoid Units for Bernoulli Output Distributions

Recalling that $J(\theta) = -E \log P(y|x)$ and $z = \omega^T h + b$

Construct an unnormalized probability distribution \tilde{P} and then normalize it

$$\log \tilde{P}(y) = yz \quad \tilde{P}(y) = \exp(yz) \quad P(y) = \frac{\exp(yz)}{\sum_{y=0}^1 \exp(y'z)} = \frac{\exp(yz)}{1 + \exp(z)}$$

that is equivalent to:

$$P(y) = \sigma((2y - 1)z)$$

z is called **logit**

If we use maximum likelihood learning,

$$J(\theta) = -E \log P(y|x) = -\log \sigma((2y - 1)z)$$

the gradient based optimization can be used (the log function undoes the exp avoiding the saturation).

$$P(y) = \frac{\exp(yz)}{1 + \exp(z)} \quad \text{is equivalent to} \quad P(y) = \sigma((2y - 1)z)$$

We have two cases:

$$y=0 \rightarrow P(0) = \frac{\exp(0)}{1+\exp(z)} = \frac{1}{1+\exp(z)} = \sigma(-z)$$

$$y=1 \rightarrow P(1) = \frac{\exp(z)}{1+\exp(z)} = \frac{1}{1+\exp(-z)} = \sigma(z)$$

Softmax Units for Multinoulli Output Distribution

L'obiettivo è quello di rappresentare una distribuzione di probabilità di una variabile discreta che ha n possibili valori.

$$\hat{y}_i = P(y = i | x) \quad i = 1..n$$

First a linear layer predicts unnormalized log probabilities:

$$z = W^T h + b \quad z_i = \log \tilde{P}(y = i | x)$$

The softmax can then exponentiate and normalize

$$\text{Soft max}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^n \exp(z_j)}$$

The likelihood function to be maximized is

$$\log \text{soft max}(z)_i = z_i - \log \sum_j \exp(z_j)$$

$$\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$

When maximizing the first term pushes z_i up while the second term pushes all the z down and can be approximated by $\max_j z_j$

Therefore the negative log likelihood cost strongly penalizes the most active incorrect prediction and this will lead to

$$\text{softmax}(z(x; \theta))_i \approx \frac{\sum_{j=1}^m 1_{y^{(j)}=i, x^{(j)}=x}}{\sum_{j=1}^m 1_{x^{(j)}=x}} \quad (\text{fraction of counts of each outcome})$$

Agli estremi, diventa una forma di "winner takes all". Con altre loss functions (che non invertono l'esponente, come MSE), questi valori sono approssimati e possiamo avere un problema di saturazione. Una versione più stabile è:

$$\text{softmax}(z) = \text{softmax}(z - \max_i z_i)$$

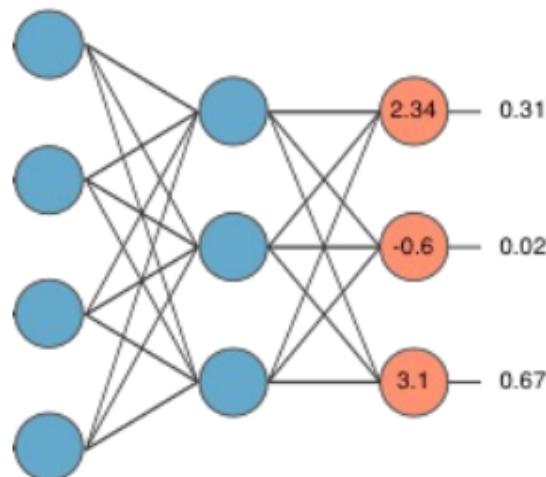
La funzione softmax è una buona funzione di output perchè è continua e differenziabile.

Softmax units for Multinoulli Output Distribution: winner takes all

Per esempio, abbiamo una rete neurale che dobbiamo allenare per classificare un'immagine tra un gatto, un cane e un essere umano.

I valori in questi nodi sono numeri reali e non hanno restrizioni.

Prima o poi ci deve essere una predizione, quindi la funzione softmax mappa i valori dei nodi con una valore tra 0 e 1, con la somma di tutti i componenti output che fa 1. Il valore più grande sarà quindi selezionato come predizione.



Since the softmax function doesn't change the ordering of the output values, the largest value before will still be the largest value after the normalization.

Quindi perché usiamo una funzione di attivazione come questa, che mantiene l'ordine, se abbiamo già abbastanza informazione per fare una predizione? Cosa succede dopo che facciamo la predizione quando la predizione è corretta, e quando è sbagliata?

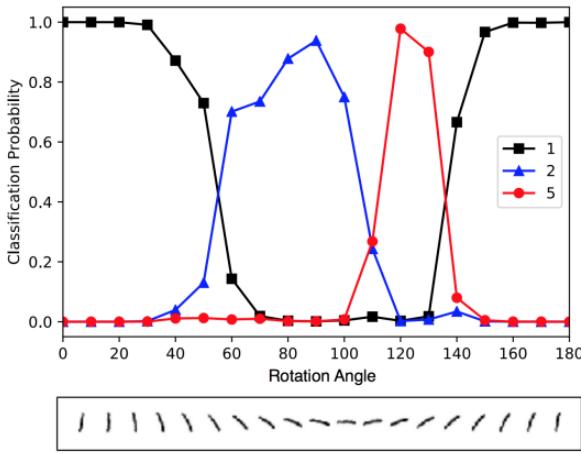
Se la loss function usa valori reali nella predizione, allora la penalità per la predizione potrebbe dover stare in un range particolare. Anche se la predizione è corretta, ci potrebbe ancora essere una penalità associata con un output di 0.75, rispetto ad uno di 0.95.

Supponiamo che ora abbiamo un classificatore addestrato. Gli facciamo vedere una foto di un gatto, e questo predice che è un gatto con un output di 0.99.

Questo significa che la foto è di un gatto con una confidenza del 99%? Cosa succede se gli facciamo vedere la foto di una tigre o di un cavallo? L'output della rete dirà che la foto del cavallo è un cane con 0.65, quindi c'è una possibilità del 65% che è un cane?

Consider now a classifier trained on the normal MNIST dataset. The classifier has only seen images of digits between 0 and 9.

What happens if we start to rotate the digits? Will the network still recognize the digit 1 if we rotate it?



As we start to rotate the digit 1, the highest output component shifts between a 1, 2, 5, and finally back to a 1 as the rotation angle increases.

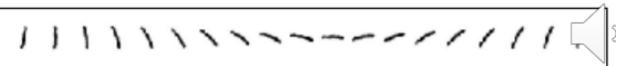
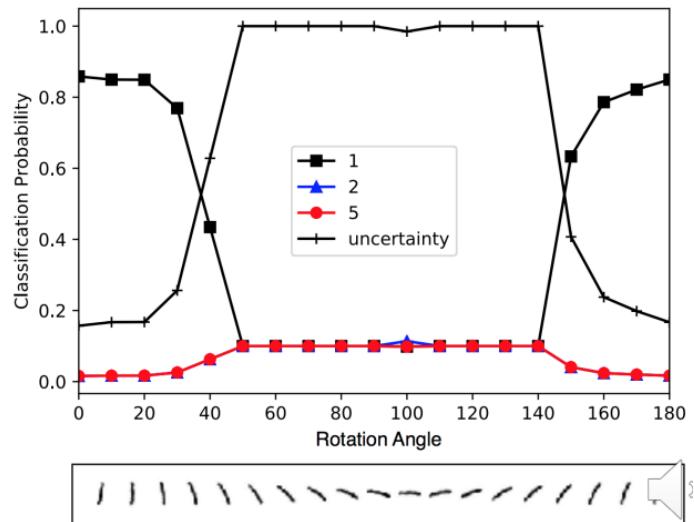
At some points, the classifier has classification probability of nearly 100% for an incorrect guess!



Solution

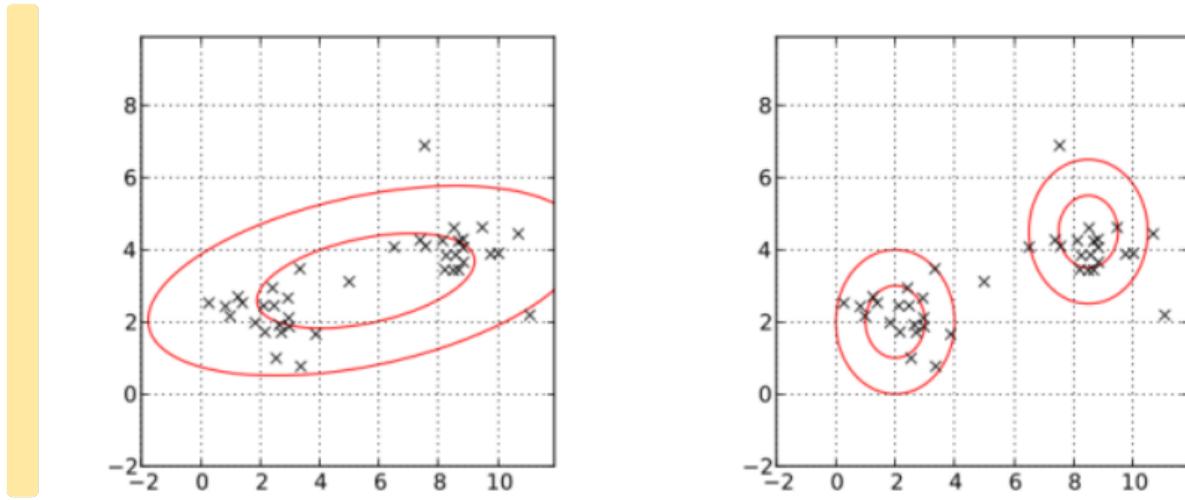
For classifiers that may have out-of-distribution samples during testing, consider adding a new option for the output: "*I don't know.*"

When an uncertainty option is used, the rotated MNIST digit makes a lot more sense



Mixture of Gaussian Output Distribution

Spesso vogliamo fare delle regressioni multimodali per predire valori reali che arrivano da una distribuzione condizionale $p(p|x)$ che ha più di un picco in y per lo stesso valore di x .



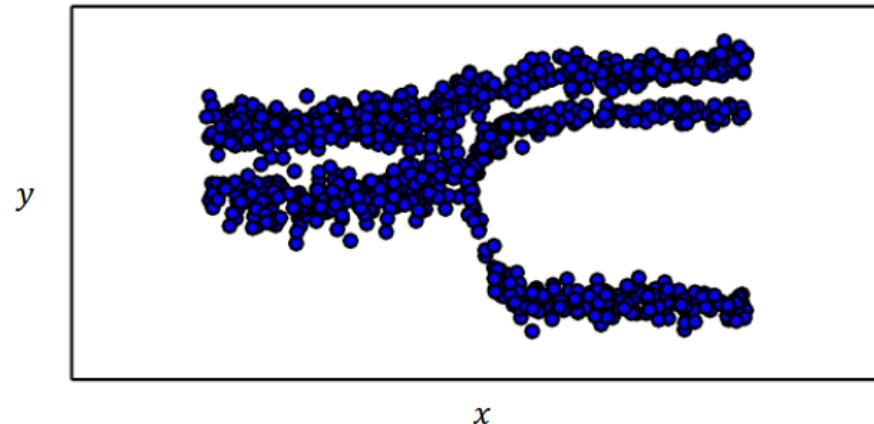
Then we can use a Gaussian Mixture

$$P(y|x) = \sum_{i=1}^n p(c = i|x)N(y; \mu^{(i)}(x), \Sigma^{(i)}(x))$$

The NN must have three outputs:

- The mixture component $p(c = i | x)$
- The means $\mu^{(i)}(x)$ and the covariances $\Sigma^{(i)}(x)$

A sample drawn from a NN with a mixture density output layer



The input x is sampled from a uniform distribution

The output y is sampled from $p_{model}(y|x)$, that is a three mixture components