Machine Learning

Neural Networks and Deep Learning: Regularization & Optimization

Gabriele Russo Russo   Francesco Lo Presti

Laurea Magistrale in Ingegneria Informatica – A.Y. 2023/24

7/11/2023

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria Civile e Ingegneria Informatica

abbiamo parlato del training, che necessita sia di regolarizzazione sia di ottimizzazione.

Regularization

Regularization

- Central problem in ML: how to make an algorithm perform well not just on training data, but also on new inputs?
- Regularization: strategies explicitly designed to reduce the test error (i.e., improve generalization), possibly at the expense of increased training error
- Many forms of regularization
  - e.g., extra constraints on the model (possibly coming from prior knowledge, or just to express generic preference for simpler model classes)
  - e.g., additional terms in the objective function f.costo + termine regolarizzazione
  - e.g., ensemble methods uso `n` modelli, aggrego le predizioni. Magari uno soffre overfitting, altri no
- We aim to trade increased bias for reduced variance

Parameter Norm Penalties

- Common regularization approach: adding a penalty $\Omega(\boldsymbol{\theta})$ to the objective function J

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \overset{\text{L loss}}{\cancel{J}}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \lambda\Omega(\boldsymbol{\theta}) \tag{1}$$

where $\lambda \in (0, \infty)$ is a hyperparameter that weighs the relative importance of the penalty term.

più lambda cresce, piu aumenta l'impatto della penalità.

Parameter Norm Penalties

- Common regularization approach: adding a penalty $\Omega(\boldsymbol{\theta})$ to the objective function J

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \lambda\Omega(\boldsymbol{\theta}) \quad (1)$$

  where $\lambda \in (0, \infty)$ is a hyperparameter that weighs the relative importance of the penalty term.
- Usually, only applied to the connection weights (i.e., $\Omega(\boldsymbol{\theta}) = \Omega(\boldsymbol{w})$), where $\boldsymbol{w}$ indicates all the weights NO bias
  - Regularizing biases doesn't give much generalization benefit and can lead to significant underfitting

Bias non si regola, penalizzarlo non dà benefici. Regolarizzo solo `w`

Parameter Norm Penalties

- Common regularization approach: adding a penalty $\Omega(\boldsymbol{\theta})$ to the objective function J

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \lambda \Omega(\boldsymbol{\theta}) \tag{1}$$

  where $\lambda \in (0, \infty)$ is a hyperparameter that weighs the relative importance of the penalty term.
- Usually, only applied to the connection weights (i.e., $\Omega(\boldsymbol{\theta}) = \Omega(\boldsymbol{w})$), where $\boldsymbol{w}$ indicates all the weights
  - Regularizing biases doesn't give much generalization benefit and can lead to significant underfitting
- Sometimes, each layer has a specific coefficient $\lambda^{(i)}$; this increases the number of hyperparameters though

Parameter Norm Penalties (2)

- The most popular choice for the norm penalty is the L^2 norm

vogliamo valori piccoli per i parametri `w` (coeff. piccoli)

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \| \boldsymbol{w} \|_2^2$$

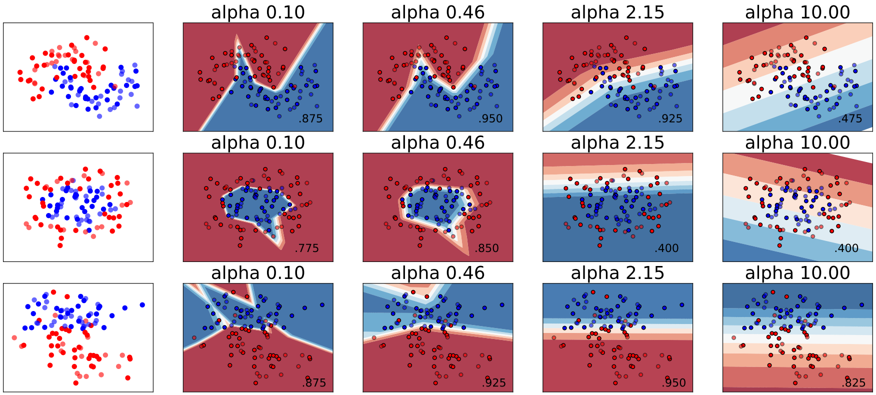
- An alternative choice is the L^1 norm

$$\Omega(\boldsymbol{\theta}) = \| \boldsymbol{w} \|_1$$

il meglio sarebbero pesi 0, ma allora Loss sarà grande, tuttavia l'algoritmo tenta di mettere a 0 più pesi possibili, ma non per forza tutti. Incentivo l'algoritmo di training che, se alcuni neuroni non in relazione con quelli del livello successivo, allora la connessione posso metterla a 0, cioè cercando rete più semplice possibile.

Example

per tre dataset diversi, classificazione binaria, impatto di avere diversi valori per il coefficiente del termine di regolarizzazione alpha (il nostro lambda). Valore piccolo, impara PERFETTAMENTE i dati in input, aumentando la divisione diventa semplice, pure troppo! Alpha = 0.46 sembra ideale.



(overfitting)

`https://scikit-learn.org/stable/auto_examples/neural_networks/plot_mlp_alpha.html#sphx-glr-auto-examples-neural-networks-plot-mlp-alpha-py`

L1/L2 Regularizers in TF

- Regularization terms can be specified per layer
- `kernel_regularizer` affects weights
- `bias_regularizer` affects bias terms

```
from keras import regularizers

layer = layers.Dense(5,
    kernel_regularizer=regularizers.L1(0.01),
    bias_regularizer=regularizers.L2(0.01))
```

norma L1 con coeff lambda = 0.01

Dataset Augmentation

- The best way to make a model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited...
- Sometimes we are able to create fake data and add it to the training set
- Whether it is possible/easy depends on the task
- Example: in object recognition, we may generate new input images by rotating/translating the existing ones

un rischio potrebbe essere abusare di questa tecnica, per forzare l'algoritmo a identificare cose a scapito di altre.

Noise Injection

- To better generalize, we would like to have NNs that are robust with respect to noise in the input
- As a form of regularization, it can be useful to artificially inject noise when training the NN
  - To the inputs (can be seen as a form of dataset augmentation!)
  - To the weights (e.g., `keras.layers.NoisyDense`)
  - To the hidden units (e.g., `keras.layers.GaussianNoise`)

applica il rumore quando serve

calcolo "h", mando al livello successivo "h+c"

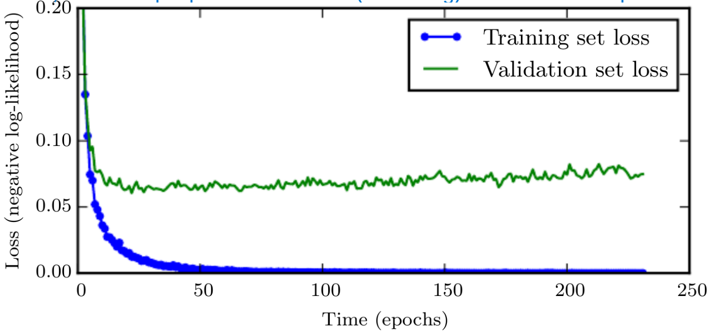
abituare la rete al fatto che ci sia del rumore nell'input.
Esempio, in una img, scurisco un po' i pixel su dei numeri, li altero leggermente.
Simile al dataset augmentation, cioè modifico in maniera casuale.

Early Stopping

da usare quasi sempre

- When training large NNs, the model has usually sufficient capacity to overfit the task
- Consider what can happen during training:

La loss deve sempre scendere sul training, fino a convergere.
Sul validation set, scende e poi potrebbe rialzarsi (overfitting). Allora mi fermo prima!



Idea: pick the parameters with lowest validation error

Early Stopping meta-algorithm

```
Data: $n$: steps between evaluations
Data: $P$: times validation error worsens before giving up
Data: $\theta_0$: initial parameters
1  $\theta \leftarrow \theta_0, \theta^* \leftarrow \theta$
2  $i, i^*, j \leftarrow 0, v \leftarrow \infty$
3  while $j < P$ do
4      Update $\theta$ running the training alg. for $n$ steps
5      $i \leftarrow i + n, j \leftarrow j + 1$
6      $v' \leftarrow \text{ValidationError}(\theta)$
7      if $v' < v$ then
8          $j \leftarrow 0$
9          $\theta^* \leftarrow \theta$
10         $i^* \leftarrow i$
11         $v \leftarrow v'$
12     end
13 end
```

quante volte massimo può vedere che l'errore cresce prima di fermarsi (perchè potrebbe crescere e poi subito decrescere)

teta sono quelli della rete
teta* sono i "migliori" trovati

j è n° volte che ho visto l'errore salire

addresto, ogni tanto mi fermo e calcolo errore validation set, tengo traccia miglior configurazione rispetto questo validation. Se vedo che P volte ho peggiorato questi risultati, mi fermo e restituisco i migliori che ho trovato durante l'iterazione, quelli prima di peggiorare.

Early Stopping (3)

non mi dice nulla sulle epoche, però se ne metto tante, l'algoritmo si ferma prima, ad esempio dopo 100, e quindi mi sta indirettamente dicendo quante epoche usare.

- Likely the most popular form of regularization in DL
  - It can also be used in conjunction with other techniques
- You can view the "training time" as a hyperparameter, which is automatically tuned through early stopping
- In principle, you also enjoy a computational benefit, as the training procedure is shortened
- However, there is the extra cost of repeatedly evaluating the validation error
  - Could be done in parallel with training on a different processor, machine, ...
- Memory overhead due to storage of multiple parameters

(due copie dei parametri mantenute)

Aggiunge overhead: prendo ogni tanto validation set e calcolo l'errore.

Early Stopping (4)

- Early stopping requires the creation of a validation set, which reduces the amount of data available for training
- To exploit the extra data, one may want to perform extra training *after* initial training with early stopping
- Common approach: re-train from scratch with whole dataset, for the number of steps determined by early stopping
- Alternative (more challenging): keep the best parameters and continue training for some (how many?!) steps with all the data

cioè con early stopping trovo numero top di iterazione, e poi dovo aver trovato questo numero alleno su tutto, però sto addestrando due volte tutto, può richiedere molto tempo.

Early Stopping in TensorFlow

funzioni richiamate durante il training

- Built-in callback for early stopping
  - monitor: metrics to monitor (e.g., `'loss'`, `'val_loss'`)
  - min_delta: minimum change to qualify as an improvement
  - patience: allowed epochs with no improvement

```
callback = keras.callbacks.EarlyStopping(\
         monitor='val_loss', patience=3)
# ...
history = model.fit(..., epochs=10,
              batch_size=16,
              callbacks=[callback])
```

Bagging

- Bagging reduces generalization error combining several models
- The techniques you studied in the first part of the course can be applied to NNs as well
- NNs reach a wide enough variety of solution points that you can often train several models on the same data with minimal changes (e.g., initial weights, minibatch selection, ...)

Dropout

- Dropout is a computationally inexpensive yet powerful method to regularize a broad family of models
  - *Srivastava, Hinton, et al., "Dropout: a simple way to prevent neural networks from overfitting", 2014*
- Can be regarded as an efficient approximation of bagging with exponentially many NNs
- Key idea: dropping out some of the units in the NN randomly
- A different sampled NN is used for every training step

Dropout (2)

- Let $p^{(\ell)} \in (0,1)$ the dropout probability for layer ℓ
  - usually 0.8 for input layer, and 0.5 for hidden units
- Let $\boldsymbol{h}^{(\ell)}$ the output of the ℓ-th layer (where $\boldsymbol{h}^{(0)} = \boldsymbol{x}$)
- For the i-th unit in layers 0, 1, ..., $(\ell - 1)$, at every training step we have:

$$\tilde{h}_i^{(\ell)} = \begin{cases} h_i^{(\ell)} & \text{with probability } p \\ 0 & \text{with probability } (1-p) \end{cases}$$

- It's like using a different NN (with shared parameters) for every example in the training set
- After training, we can use the NN *without* dropout, scaling the weights: $\boldsymbol{W}_{test}^{(\ell)} = p^{(\ell)} \boldsymbol{W}^{(\ell)}$

Dropout in TensorFlow

- Dropout available as a layer in Keras
- The Dropout layer randomly sets its **input** units to 0 with a frequency of `rate` at each step **during training** (
- Inputs not set to 0 are scaled up by `1/(1 - rate)` such that the sum over all inputs is unchanged
  - Keras performs input scaling during training as well
  - Slightly different than scaling weights only after training (as described before), but equivalent in practice

```
layer = tf.keras.layers.Dropout(0.5,
                    input_shape=(2,))
```

Optimization Algorithms for NNs

Training

- ML largely relies on mathematical optimization and, especially, gradient-based optimization
- Training large NNs is a challenging task, which can take days or months using 100+ machines
- It is not surprising that specialized optimization techniques have been studied for this problem
- We will briefly review some of these techniques
    - note that an exhaustive and rigorous discussion of the optimization issues in ML is out of the scope of the lectures

Learning vs. Pure Optimization

- Important differences between ML and pure optimization
- The main one: ML acts indirectly
- In pure optimization, given an objective J, you try to minimize J and evaluate a solution based on the value of J
- In ML, you care about a performance measure P defined w.r.t. the test set and possibly intractable
- Therefore, you optimize a different cost function J w.r.t. the training set, hoping that doing so will improve P
- Furthermore, ML training does not stop at (local) minima, but usually when a convergence criterion is met (e.g., early stopping)

Challenges in NN Optimization

- Optimization is difficult; many ML models have been carefully designed to deal with convex optimization problems (which can be still challenging!)
  - Finding a local minimum is enough (it will be a global minimum)
- When training NNs, we must confront the general nonconvex case

Local Minima and Saddle Points

- Deep models guaranteed to have an extremely large number of local minima
- Problematic if they have cost higher than global minimum
- So, are local minima a major problem in NN training?
  - Still an open research question (difficult to establish in high-dimensional spaces)
  - Today, we suspect that, for large NNs, most local minima have a low cost function value

Local Minima and Saddle Points

- Deep models guaranteed to have an extremely large number of local minima
- Problematic if they have cost higher than global minimum
- So, are local minima a major problem in NN training?
  - Still an open research question (difficult to establish in high-dimensional spaces)
  - Today, we suspect that, for large NNs, most local minima have a low cost function value
- For high-dimensional nonconvex functions, more saddle points than local minima
  - Local minimum *and* maximum along different cross-sections
  - Zero gradient, and very small around the point
  - SGD seems quite good at escaping

Vanishing and Exploding Gradients

- Suppose that a computational graph contains a path where a matrix $\boldsymbol{W}$ is repeatedly multiplied
- After t steps, it is equivalent to $\boldsymbol{W}^t$ and for eigendecomposition:

$$\boldsymbol{W}^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}$$

- Any eigenvalue λ_i with $|\lambda_i|$ not close to 1 will either vanish or explode, and so the gradients along this graph

Vanishing and Exploding Gradients

- Suppose that a computational graph contains a path where a matrix $\boldsymbol{W}$ is repeatedly multiplied
- After t steps, it is equivalent to $\boldsymbol{W}^t$ and for eigendecomposition:

$$\boldsymbol{W}^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}$$

- Any eigenvalue λ_i with $|\lambda_i|$ not close to 1 will either vanish or explode, and so the gradients along this graph
- Mostly regards recurrent NNs, which repeatedly apply the same operation
- Feedforward DNNs use different weights at each layer and can largely avoid the issue

Algorithms

We will see a few algorithms commonly used to train NNs.
We will start from the one you already know, i.e., SGD.

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) at epoch *k*

```
1 while stopping criterion not met do
2     Sample minibatch $\mathcal{B}$ randomly from the training set
3     $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
4     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha_k \boldsymbol{g}$
5 end
```

- The learning rate is a crucial paramter for SGD
- So far, we always considered a constant learning rate α
- Let α_k the learning rate at epoch k

SGD: Learning rate

- It is necessary to gradually decrease α_k over time
  - The random selection of samples in SGD introduces noise in gradient estimation, even when we arrive at minimum
- Sufficient conditions for convergence (both must hold):

$$\sum_{k=1}^{\infty} \alpha_k = \infty \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

- Common approach: linearly decaying learning rate

$$\alpha_k = \begin{cases} \alpha_0 - \frac{k}{\tau}(\alpha_0 - \alpha_\tau) & k \leq \tau \\ \alpha_\tau & k > \tau \end{cases}$$

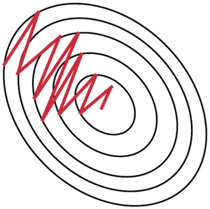
- How to set α_0, α_τ, τ? More of an art than a science…
  - Usually $\alpha_\tau \approx 0.01\alpha_0$
  - Best to look at learning curves

Example: Learning Rate Schedule

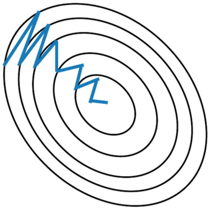
```
s = keras.optimizers\
        .schedules.ExponentialDecay(
    initial_learning_rate=1e-2,
    decay_steps=10000,
    decay_rate=0.9)
opt = keras.optimizers.SGD(learning_rate=s)
```

Momentum

- SGD is a popular choice, but it can be very slow to converge
- The method of momentum aims to accelerate learning
- Idea: accumulate an exponentially decaying moving average of past gradients and continue to move in that direction



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

Momentum (2)

- Momentum comes from a physics analogy, in which the negative gradient is a force moving a particle through parameter space
- A variable $\boldsymbol{v}$ plays the role of velocity
  - Actually, in physics, momentum is mass times velocity (with unit mass, $\boldsymbol{v}$ represents momentum)
- A hyperparameter $\beta \in [0, 1)$ determines how fast past contributions decay

Momentum (3)

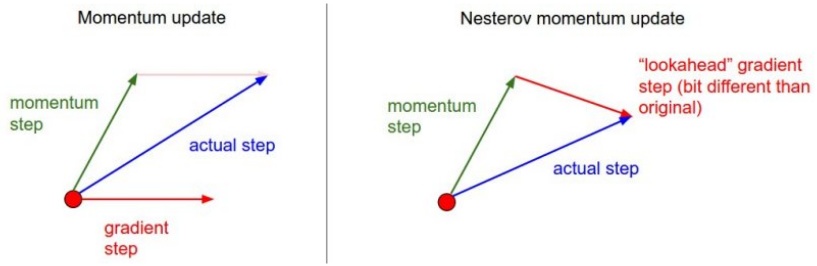
Stochastic Gradient Descent (SGD) with momentum

```
while stopping criterion not met do
    Sample minibatch $\mathcal{B}$ randomly from the training set
    $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
    Update velocity: $\boldsymbol{v} \leftarrow \beta \boldsymbol{v} - \alpha_k \boldsymbol{g}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end
```

- SGD step size depends on length and “alignment” of gradient sequence
- Common values for β: 0.5, 0.9, 0.99
- Keras: `SGD(momentum=beta,...)`:

Nesterov Momentum

- Variant of the momentum algorithm
- Adds a correction by evaluating gradient *after* applying the current velocity



Nesterov Momentum (2)

Stochastic Gradient Descent (SGD) with Nesterov momentum

```
1 while stopping criterion not met do
2     Sample minibatch $\mathcal{B}$ randomly from the training set
3     Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \beta \boldsymbol{v}$
4     $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J(\tilde{\boldsymbol{\theta}})$
5     Update velocity: $\boldsymbol{v} \leftarrow \beta \boldsymbol{v} - \alpha_k \boldsymbol{g}$
6     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
7 end
```

- Keras: `SGD(momentum=beta, nesterov=True, ...)`[1]

[1]The actual implementation is an approximation of Nesterov Momentum, see: `https://stackoverflow.com/a/50778921`

More Algorithms…

- Learning rate is one of the most difficult to set hyperparameters
- Momentum can mitigate some issues, but introduces another hyperparameter
- Idea: using a separate learning rate for each parameter and automatically adapt them

AdaGrad

- Adapt individual learning rate of all model params
- Scaling inversely proportional to square root of the sum of all the historical squared values of the gradient
- “Larger derivatives lead to more rapid decrease”
- AdaGrad has good theoretical properties for convex settings
- Difficult to use in practice due to gradient storage requirement
  - Still worth presenting because of its extensions!

AdaGrad: Algorithm

AdaGrad

```
$\boldsymbol{r} \leftarrow 0$                    /* Gradient accumulation */
while stopping criterion not met do
    Sample minibatch $\mathcal{B}$ randomly from the training set
    $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
    $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + -\frac{\alpha}{\delta+\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
end
```

Notes:

- $\odot$ denotes element-wise operations
- δ is a small positive constant (usually 10^{-7})

RMSProp

- Proposed by Hinton in 2012, RMSProp (Root Mean Squared Propagation) modifies AdaGrad
- One of the most used optimization algorithms for DNNs
- Gradient accumulation becomes an exponentially moving average
  - Nonconvex problems better handled
  - No need to store all the gradients
- New hyperparameter ρ: length scale of the moving average

RMSProp: Algorithm

RMSProp

```
r ← 0                                        /* Gradient accumulation */
while stopping criterion not met do
    Sample minibatch $\mathcal{B}$ randomly from the training set
    $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
    $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + -\frac{\alpha}{\delta+\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
end
```

Notes:

- $\odot$ denotes element-wise operations
- δ is a small positive constant (usually 10^{-6})

RMSProp: Algorithm (2)

RMSProp with Nesterov momentum

```
1 $\boldsymbol{r} \leftarrow 0$                    /* Gradient accumulation */
2 **while** *stopping criterion not met* **do**
3     Sample minibatch $\mathcal{B}$ randomly from the training set
4     Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \beta \boldsymbol{v}$
5     $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}|} \nabla_{\boldsymbol{\theta}} J(\tilde{\boldsymbol{\theta}})$
6     $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
7     $\boldsymbol{v} \leftarrow \beta \boldsymbol{v} - \frac{\alpha}{\sqrt{r}} \odot \boldsymbol{g}$
8     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
9 **end**
```

Adam

- Proposed in 2014, Adam ("adaptive moments") can be seen as a variant of RMSProp with momentum
- Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients
- Important differences w.r.t. RMSProp:
  - Momentum directly incorporated as an estimate of the first-order moment of the gradient (while in RMSProp is applied to rescaled gradient, without theoretical justification)
  - Estimates of the first- and second-order moments are corrected to avoid initialization bias (moving averages are initialized as 0)

Choosing the Algorithm

- How to choose the algorithm to use?
- No single answer!

- The algorithms we have discussed are popular choices
- User's familiarity with the algorithm is an important factor in the choice
  - If you are not familiar, difficult to tune hyperparameters

Parameter Initialization

- How parameters are initialized determines the initial point (in the parameter space) where the optimization starts
- The initial point can determine whether the algorithm converges at all!
  - some initial points are so unstable that the algorithm encounters numerical difficulties
- When learning does converge, the initial point can determine converge speed and whether a point with high or low cost is reached
- Various heuristics proposed, usually random-based

Parameter Initialization (2)

- Random Normal/Uniform: sample weights randomly from a given distribution
- Xavier[2] method: now almost a standard practice
  - weights randomly sampled from a Gaussian distribution
  - $\mu = 0$
  - $\sigma^2 = \frac{2}{n_{IN}+n_{OUT}}$, where n_{IN} and n_{OUT} denote the number of inputs and outputs in the layer

```
init = keras.initializers.GlorotNormal()
layer = keras.layers.Dense(3,
          kernel_initializer=init)
```

[2]Named after Xavier Glorot, who proposed the method with Bengio

Batch Normalization

- Batch normalization (BN) is a popular and effective technique to accelerate the convergence of deep NNs
  - Not an optimization algorithm, but a tool to improve convergence
  - Regularization effect as a secondary benefit
  - Proposed in 2015
- BN is applied to individual layers (or, possibly, to all of them)
- Input re-scaling and standardization benefits optimizers, since it puts the parameters on a similar scale
- It is natural to ask whether a similar normalization step inside a NN might be beneficial

Batch Normalization (2)

- Consider a minibatch $\mathcal{B}$ and an input $\boldsymbol{x} \in \mathcal{B}$

$$BN(\boldsymbol{x}) = \frac{\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}}$$

  where $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ and $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ denote the mean and std. deviation of the minibatch. (Note: they are vectors with the same shape of $\boldsymbol{x}$)
- The resulting minibatch has zero mean and unit variance

Batch Normalization (3)

- In practice, BN is applied to hidden layers

$\boldsymbol{h} = \phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ becomes $\boldsymbol{h} = \phi(BN(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}))$

- The expression above follows the original paper presenting BN; actually, implementations often apply BN *after* the activation function

$$\boldsymbol{h} = BN(\phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}))$$

Batch Normalization (4)

- Layer activations seem to be limited in their expressive power (e.g., they are forced to have unit variance)
- A more general BN formulation:

$$BN(\boldsymbol{x}) = \gamma \frac{\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}$$

  where the *vectors* γ and $\boldsymbol{\beta}$ are parameters to learn as part of training

Batch Normalization (4)

- Layer activations seem to be limited in their expressive power (e.g., they are forced to have unit variance)
- A more general BN formulation:

$$BN(\boldsymbol{x}) = \gamma \frac{\boldsymbol{x} - \hat{\mu}_{\mathcal{B}}}{\hat{\sigma}_{\mathcal{B}}} + \boldsymbol{\beta}$$

  where the *vectors* γ and $\boldsymbol{\beta}$ are parameters to learn as part of training
- Question: why did we force the mean to be zero and now introduce a parameter to allow for any mean??
  - We can represent the same family of functions as before, but now we have different (and easier!) learning dynamics
  - Mean and variance solely depend on γ and $\boldsymbol{\beta}$, whilst without BN they depend on the complicated interaction of the parameters in all the previous layers!

Batch Normalization (5)

- The formulas presented so far describe how BN works during training
- The behavior is slightly different at inference time, because it must be able to work without minibatches (e.g., prediction against a single input value)
  - Question: what happens if you apply BN as described so far to a minibatch of size 1?
- At inference time, normalization uses a moving average of the mean and standard deviation of the minibatches seen during training

Batch Normalization in Keras

- It is extremely easy to integrate BN in Keras
- It is offered as specific type of layer, to be used *after* the hidden layer where it must be applied:

```
model = keras.Sequential()
model.add(layers.Dense(16))
model.add(layers.BatchNormalization())
```