Feed Forward Neural Networks
Fancy Stuff in Deep Learning

Contents

1 Regularization

One possible definition of *regularization* in machine learning is : "*any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error*". In machine learning in general, there exist many different strategies. In deep learning almost all rely on the regularization of an estimator (in the bias-variance minimization tradeoff sense).

1.1 Parameter norm penalty

The easiest and mostly used regularization method, even outside of deep learning is parameter norm penalty. Given an objective function $J(\cdot)$ over a dataset $\{X, \boldsymbol{t}\}$ and a parameter θ, we introduce a regularized cost :

$$\tilde{J}(\theta, X, \boldsymbol{t}) = J(\theta, X, \boldsymbol{t}) + \alpha\Omega(\theta) \tag{1}$$

with $\alpha > 0$ weighting the contribution of the norm-penalty term $\Omega(\cdot)$.

Remark : For deep learning norm penalization is applied only on weights, and not on biases. Think about the penalty as the gradient norm reduction

There exist many classical norm penalizations, widely used in machine learning:

- $\mathcal{L}^2$ regularization (or Ridge or Tikhonov) :

$$\begin{aligned}\Omega(\theta) &= \frac{1}{2}\theta^T\theta \\ &= \frac{1}{2}\|\theta\|_2^2\end{aligned} \tag{2}$$

The optimization algorithm will perceive the input X as having higher variance (write the solution given by the normal equations when having a sum-of-squared errors objective function).

- $\mathcal{L}^1$ regularization (or LASSO) :

$$\begin{aligned}\Omega(\theta) &= \|\theta\|_1 \\ &= \sum_{i=1}^{n} |\theta_i|\end{aligned} \tag{3}$$

- Direct norm constrained optimization

$$\begin{aligned} \min_{\theta} \quad & J(\theta, X, \boldsymbol{t}) \\ \text{s.t.} \quad & \Omega(\theta) \leq k \end{aligned} \tag{4}$$

  with an hard-constraint on the parameter.

1.2 Noise robustness

For linear models with Gaussian noise, the addition of noise with infinitesimal variance on the input of the model is equivalent to imposing a norm penalty on the weights.

Another possibility is to apply noise on the weights - which can be interpreted as a naive stochastic implementation of Bayesian inference. It also encourages stability. Indeed, in a regression setting, we'll optimize the weights over

$$J = \mathbb{E}_{p(x,y)} \left[\left(\hat{y}(x) - y\right)^2 \right] \tag{5}$$

with $\hat{y}(x)$ being the current prediction. If we include a random perturbation $\varepsilon_W \sim \mathcal{N}(\varepsilon \mid 0, \eta \mathbb{1})$ to the networks weights and denote $\hat{y}_{\varepsilon_W}(x)$ the resulting perturbed prediction, the corresponding objectives now writes :

$$\begin{aligned}\tilde{J}_W &= \mathbb{E}_{p(x,y,\varepsilon_W)} \left[\left(\hat{y}_{\varepsilon_W}(x) - y\right)^2 \right] \\ &= \mathbb{E}_{p(x,y,\varepsilon_W)} \left[\hat{y}_{\varepsilon_W}(x)^2 + y^2 - 2y\hat{y}_{\varepsilon_W}(x) \right]\end{aligned} \tag{6}$$

If $\eta \ll 1$, minimizing $\tilde{J}_W$ is therefore equivalent to minimizing J with an additional regularization term :

$$\boxed{\Omega(W) = \eta \mathbb{E}_{p(x,y)} \left[\|\nabla_W \hat{y}(x)\|^2 \right]} \tag{7}$$

This encourages the parameters to *lie in regions of parameter space where small perturbations of the weights have a relatively small influence* in the output. We are therefore looking for a minima with large, flat surrounding regions.

1.3 Early stopping

When training a model with sufficient representational capacity to overfit, we see a steady decrease in the training error along the learning. The test error follows that behavior until it starts increasing again : this is a good proof that the network is starting to overfit.

One strategy known as early stopping is to store the parameters value every time an improvement on the validation set is met. We return such parameters when the training is over. This strategy has several advantages :

- It is effective and simple
- It induces no changes to the initial training procedure
- It allows to determine an optimal number of training steps

It is also possible to show that it is equivalent to $\mathcal{L}^2$ regularization in a linear regression scheme.

1.4 Dropout

This method can be seen as making bagging (or *bootstrap aggregating*) practical for ensembles of many large neural networks (which can be hard when models are very deep nets !). Dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from an underlying base network.

Each time we load an exemple into a minibatch, we randomly sample a different binary mask to apply to all the hidden and input units in the networks (the probability, for a node, of being selected is an hyper-parameter determined by the user). We then run forward-propagation, back-propagation and the learning update using the newly obtained topology.

More formally, let μ be a mask specifying which units to include and $J(\theta, \mu)$ defining the cost of the model defined by parameter θ and mask μ. We wish to minimize the joint-distribution expectation :

$$\mathbb{E}_\mu \left[J(\theta, \mu) \right] \tag{8}$$

by sampling values of μ from its specified distribution. Now, if we were to make a prediction, we would accumulate votes from all member, and therefore compute :

$$\hat{y}_\infty(x) = \sum_u p(\mu) \mathbb{E} \left[y | \mu, x \right] \tag{9}$$

Because this involves summing over 2^d possibilities (where d are the number of units that can be dropped), it is of course intractable. A key insight involved in dropout is that we can approximate the committees vote by evaluating $p(y \,|\, x)$ in one model : the one with *all-units*, but with the weights going out of unit i multiplied by the probability of including unit i. Usually, dropout is performed with a uniform inclusion probability of $1/2$. Then, we simply have to half all the weights before making predictions.

Remark : This rule is actually exact for many classes of model that don't have non-linear activation functions.

1.5 Others

There exist of course many other regularization strategies for deep neural networks (direct bagging, weight-sharing, dataset augmentation ..) that we don't describe here.

2 Optimization

It is important to understand how learning differs from pure optimization. In most machine learning scenarios, we are interested in reducing some quantity $P(\cdot)$ defined with respect to the test set. We are only able to do so indirectly : we reduce a training error $J(\cdot)$, hoping to improve $P(\cdot)$. Indeed - be it in supervised or unsupervised learning - we would like to optimzie the mean of the loss-function over the real data-generating distribution p_{data} :

$$J^*(\theta) = \underbrace{\mathbb{E}_{(x,y)\sim p_{data}} \left[L(f(x,\theta), y) \right]}_{\text{risk}} \tag{10}$$

however that distribution is unknown ! We therefore minimize an *empirical risk* :

$$\begin{aligned} J(\theta) &= \mathbb{E}_{(x,y)\sim \tilde{p}_{data}} \left[L(f(x,\theta), y) \right] \\ &= \frac{1}{N} \sum_{n=1}^{N} L(f(x^{(n)}, \theta), y^{(n)}) \end{aligned} \tag{11}$$

which is prone to overfitting ! This is mainly why we use regularization techniques.

2.1 Minibatch optimization

Minibatch optimization is a good exemple of how learning differs from pure optimization. When we assume indepedence between samples, the likelihood takes up a factorial form :

$$\mathcal{L}\left(\boldsymbol{t} | X, \theta \right) = \prod_{n=1}^{N} \mathcal{L}(t^{(n)} | x^{(n)}, \theta) \tag{12}$$

which leads to a summative expression of the gradient of the log-likelihood :

$$\nabla_\theta \left\{ \log \mathcal{L}\left(\boldsymbol{t} | X, \theta \right) \right\} = \nabla_\theta \left\{ \sum_{n=1}^{N} \log \mathcal{L}\left(t^{(n)} | x^{(n)}, \theta \right) \right\} \tag{13}$$

Therefore, if we omit a few terms, we are given a good approximation of the gradient :

$$\nabla_\theta \left\{\log \mathcal{L}\left(\boldsymbol{t}|X,\theta\right)\right\} = \nabla_\theta \left\{\sum_{n=1}^{M} \log \mathcal{L}\left(\tilde{t}^{(n)}|\tilde{x}^{(n)},\theta\right)\right\} \tag{14}$$

where $M \leq N$ and the couples $\left(\tilde{x}^{(n)},\tilde{t}^{(n)}\right)$ are independently drawn from the training set. This reduced set is known as a *mini-batch*. Its extreme cases are known as *on-line* learning ($M = 1$) and batch learning ($M = N$).

Mini-batches size are generally driven by the following factors :

- Larger batches provide a more accurate estimate of the gradient, but with less than linear return (remember that the variance varies with $\sqrt{M}$).
- The use of multicore architecture or parallel optimization schemes set a proper reasoning rule of thumb for choosing a batch size.
- Small batches can have a regularizing effect (but need smaller learning rates because of the gradient's estimate variance - and hence require more learning steps)

Remark : It is crucial that the samples are independent in order to reduce the bias of the gradient's estimate. A good rule of thumb is to proceed to a random shuffle of the dataset if some sequences were to be correlated (same measure,..)

2.2 Challenges in neural network optimization

Deep neural network training often involves optimizing a non-convex objective. It shares many of the convex optimization challenges, and some additional ones :

- Ill-conditioning : $g^THg \gg g^Tg$ or strong curvature near a local minima
- Local minima : especially in deep learning there is a local minima problem, because of the *model identifiability problem* (weight space symmetry - although this lead to equivalent local minima).
- Others : Plateaus, saddle points, flat regions, cliffs and exploding gradients.

2.3 Basic algorithms

2.3.1 Stochastic gradient descent

Stochastic gradient descent - as we described it earlier - and its variance are probably the most used optimization algorithms for machine learning in general, and for deep-learning in particular.

Remark : As a reminder, it is possible to obtain an unbiased estimate of the gradient by taking the average gradient on a mini-batch of i.i.d samples.

Because of the noise inherent to the SGD procedure that does not vanish even at a local minimum, the learning rate must comply with the stochastic optimization conditions (15) :

$$\boxed{\begin{aligned}\sum_{k=0}^{\infty} \varepsilon_k &= +\infty \\ \sum_{k=0}^{\infty} \varepsilon_k^2 &< +\infty\end{aligned}} \tag{15}$$

for instance, with

$$\varepsilon_k = \frac{\alpha_0}{k^\beta}, \quad \beta > 0.5 \tag{16}$$

In practice, we commonly use :

$$\varepsilon_k = (1-\alpha_k)\varepsilon_0 + \alpha_k\varepsilon_\tau \text{ with } \alpha_k = \frac{k}{\tau} \tag{17}$$

and keep $\varepsilon_k = \varepsilon_\tau \, \forall k \geq \tau$.

2.3.2 Momentum

This method is designed to accelerate the learning process, especially in the case of high curvatures, small but consistent gradients or noisy gradients. It accumulates an exponentially decaying average of past gradients and continues to move in their directions.

Formally, with $\alpha \in [0, 1[$, the update rule writes :

$$\boxed{\begin{aligned} & v \leftarrow \alpha v - \varepsilon \cdot \nabla_\theta \left(\frac{1}{M} \sum_{m=1}^{M} L(f(x^{(m)}, \theta), y^{(m)}) \right) \\ & \theta \leftarrow \theta + v \end{aligned}} \tag{18}$$

with v initially set to 0. One possible use of the momentum method writes :

ALGORITHM - SGD with momentum

```
  **Input**: Learning rate $\varepsilon$, momentum parameter $\alpha$
  **Input**: Initial $\theta_0$, initial velocity $v_0$
1 **while** *stopping criterion is not met* **do**
2     Sample mini-batch of size $M$ : $\{X^{(M)}, y^{(M)}\}$ ;
3     Compute gradient estimate;
4         $g \leftarrow \nabla_\theta \left( \frac{1}{M} \sum_{m=1}^{M} L(f(x^{(m)}, \theta), y^{(m)}) \right)$
      Update velocity ;
5         $v \leftarrow \alpha v - \varepsilon g$
      Update parameter ;
6         $\theta \leftarrow \theta + v$
7 **end**
```

The size of the step now depends on how large and how aligned a sequence of gradients are. In the extreme case where the momentum algorithm always sees the same gradient g, the size of the step converges to

$$v_\infty = \frac{\varepsilon \|g\|}{1 - \alpha} \tag{19}$$

Typically, the momentum accelerates the maximum speed by a factor $\frac{1}{1-\alpha}$. α can also be tuned along the learning (set near 0 and then grow for instance).

2.3.3 Nesterov momentum

This method is inspired by Nesterov accelerated gradient method. The update rule now writes :

$$\boxed{\begin{aligned} & \theta' \leftarrow \theta + \alpha v \\ & v \leftarrow \alpha v - \varepsilon \cdot \nabla_\theta \left(\frac{1}{M} \sum_{m=1}^{M} L(f(x^{(m)}, \boldsymbol{\theta'}), y^{(m)}) \right) \\ & \theta \leftarrow \theta + v \end{aligned}} \tag{20}$$

The gradient is now evaluated after the current velocity is applied, in an attempt to add a correction factor.

ALGORITHM - SGD with Nesterov momentum

```
  Input: Learning rate $\varepsilon$, momentum parameter $\alpha$
  Input: Initial $\theta_0$, initial velocity $v_0$
1 while stopping criterion is not met do
2     Sample mini-batch of size $M$ : $\{X^{(M)}, y^{(M)}\}$ ;
3     Apply interim update;
4         $\theta' = \theta + \alpha v$
      Compute gradient estimate;
5         $g \leftarrow \nabla_\theta \left( \frac{1}{M} \sum_{m=1}^{M} L(f(x^{(m)}, \theta'), y^{(m)}) \right)$
      Update velocity ;
6         $v \leftarrow \alpha v - \varepsilon g$
      Update parameter ;
7         $\theta \leftarrow \theta + v$
8 end
```

In the convex batch gradient case, Nesterov momentum brings the rate of convergence of the excess error from $O(\frac{1}{k})$ to $O(\frac{1}{k^2})$.

2.4 Parameter initialization strategies

Such strategies are by nature simple and heuristic and are based on achieving nice properties when the network is initialized.

> **Remark** : Such properties might not be kept during the optimization process. What's more, the understanding on how the initial point affects the generalization skills of a neural network is extremely primitive.

However, one rule is certain : the initial parameters need to break symmetry between different units. This namely motivate random initialization of the weights. Still, comes the questions of range - we neither want a fading nor exploding process during forward propagation. Closer to the regularization issue, we could think of a Gaussian initialization of hte weight as a naive way to enforce a Gaussian prior $p(\theta \,|\, \theta_0, \sigma^2) = \mathcal{N}\left(\theta \,\middle|\, \theta_0, \sigma^2 \mathbb{1}\right)$. Hence there exist many diferent optimization strategies :

- Normalized initialisation
$$W \sim \mathbb{U}\left(-\sqrt{\frac{6}{m+n}}, -\sqrt{\frac{6}{m+n}}\right) \tag{21}$$
- Scaling initialization
- Sparse initialization
- ...

2.5 Adaptative learning rates algorithms

The learning is one of the most difficult parameters to set-up. The objective function we are trying to optimize is often highly sensitive to some directions end insensitive to others. The momentum algorithm can mitigate these issues, but doe so at the expense of introducing another hyper-parameter. Also, line-search algorithms can only be applied to full-batch optimization, and often come at the cost of computational burdens. In the following, we review a few number of mini-batch based methods that perform learning rate adaptation.

2.5.1 AdaGrad

This algorithm individually adapts the learning rate of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient. Therefore, the parameters with the largest partial derivatives of the loss have a rapid decrease in their learning rate.

ALGORITHM - AdaGrad with SGD

```
    Input: Learning rate $\varepsilon$, initial parameter $\theta$, small constant $\delta(\sim 10^{-7})$
1   Initialize $r = 0$;
2   while stopping criterion is not met do
3       Sample minibatch;
4       Compute gradient $g$ and the accumulated squared gradient $r \leftarrow r + g \odot g$;
5       Compute update $\Delta\theta \leftarrow -\frac{\varepsilon}{\delta+\sqrt{r}} \odot g$;
6       Update parameter $\theta \leftarrow \theta + \Delta\theta$;
7   end
```

Remark : In the context of convex optimization, AdaGrad enjoyrs some desirable theoretical properties. Empirically, the accumulation of squared gradients from the beginning of the training can result in a premature decrease in the effective learning rate.

2.5.2 RMSProp

The idea behind RMSPropr is to modify AdaGrad in a non-convex setting by changing the gradient accumulation into an exponentially moving average. Therefore, we discard history from the extreme past so that the optimization procedure can rapidly converge after finding a convex bowl.

ALGORITHM - RMSProp with Nesterov momentum

```
    Input: Learning rate $\varepsilon$, initial parameter $\theta$, decay rate $\rho$, initial velocity, small constant
           $\delta(\sim 10^{-7})$
1   Initialize $r \leftarrow 0$;
2   while stopping criterion is not met do
3       Sample minibatch;
4       Compute interim update $\theta' = \theta + \alpha v$;
5       Compute gradient $g \leftarrow \nabla_\theta \left( \frac{1}{M} \sum_{m=1}^{M} L(f(x^{(m)}, \theta'), y^{(m)}) \right)$;
6       Accumulate gradient $r \leftarrow \rho r + (1-\rho) g \odot g$;
7       Compute velocity update $v \leftarrow \alpha v - \frac{\varepsilon}{\delta+\sqrt{r}} \odot g$;
8       Apply update $\theta \leftarrow \theta + v$;
9   end
```

The hyper-parameter ρ controls the length scale of the moving average. Empirically, RMSProp has been shown to be an effective an practical optimization algorithm for deep neural networks - and is currently one of the go-to optimization algorithm by deep learning practitioners.

2.5.3 Adam

This name stands for "*Adaptative Moments*". It is best seen as a RMSProp algorithm with a momentum variant, which is incorporated directly as an estimate of the first order moment of the gradient.

ALGORITHM - RMSProp with Nesterov momentum

```
Input: Learning rate $\varepsilon$, initial parameter $\theta$, exponential decay rate s$\rho_1$ and $\rho_2$, initial
       velocity, small constant $\delta(\sim 10^{-7})$
1  Initialize $r \leftarrow 0$;
2  while stopping criterion is not met do
3      Sample minibatch;
4      $t \leftarrow t+1$;
5      Compute interim update $\theta' = \theta + \alpha v$;
6      Compute gradient $g \leftarrow \nabla_\theta \left( \frac{1}{M} \sum_{m=1}^{M} L(f(x^{(m)}, \theta'), y^{(m)}) \right)$;
7      Update biased first moment estimate $s \leftarrow \rho_1 s + (1-\rho_1) g$;
8      Update biased second moment estimate $r \leftarrow \rho_1 r + (1-\rho_1) g \odot g$;
9      Correct bias first moment : $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$;
10     Correct bias first moment : $\hat{r} \leftarrow \frac{r}{1-\rho_1^t}$;
11     Compute update : $\Delta\theta \leftarrow -\varepsilon \frac{\hat{s}}{\delta+\sqrt{\hat{r}}}$ (component-wise);
12     Apply update : $\theta \leftarrow \theta + \Delta\theta$
13 end
```

2.6 Approximate second order methods

What we previously saw are first order methods (steepest-descent like) with adaptative learning rates. Provided the Hessian of the function function at θ_k, denoted H_k, one second order optimization scheme is the Newton descend :

$$\theta_{k+1} = -H_k^{-1} \nabla_\theta J(\theta)|_{\theta_k} \tag{22}$$

This descent direction is based on a locally quadratic approximation of the objective function. In deep learning the objective function is typically non-convex (saddle points) meaning that often :

$$\exists \theta \in \Theta,\, H_k = \nabla_\theta^2 J(\theta) \notin \mathcal{S}_n^{++}(\mathbb{R}) \tag{23}$$

leading the update in the wrong direction.

To avoid this phenomenon, one can use a regularized update, known as the Levenberg-Marquardt update :

$$-\left(H_k + \alpha \mathbb{1}\right)^{-1} \nabla_\theta J(\theta)|_{\theta_k}, \quad \alpha > 0 \tag{24}$$

which is actually a mix between Newton and gradient descent.

As in many optimization practical cases, and particularly in deep learning one cannot afford to compute and invert the Hessian at every step. *Quasi-Newton* methods are designed to reduce the related computational burden by keeping an estimate of H_k or directly H_{k-1} along the learning. One such method is known as BFGS, and update M_t, an approximation of H^{-1} by adding low-rank matrixes along the optimization process.

However, the stockage of M_t is $O(n^2)$ in memory, making it impractical for most modern deep learning models. The L-BFGS (Low-Memory BFGS) answer this problem by assume M_{t-1} to be the identity matrix (of course, there exist many more sophisticated variants).