Machine Vision Practical 10: MLP cheatsheet

University College London

February 5, 2018

1 TL;DR

For the `MATLAB` practical 10a the relevant equations are in blue. CAVEAT: We use ***row-vector*** notation, so instead of writing $\mathbf{y} = \mathbf{W}\mathbf{x}$, we write $\mathbf{y} = \mathbf{x}\mathbf{W}$. This is so that the notation matches the `MATLAB`/`Python` implementation (and every other deep learning library). These can be found at

(10) `layers/relu_layer/forward`

(12) `layers/relu_layer.m/backward`

(14) `layers/affine_layer.m/forward`

(17) `layers/affine_layer.m/backward`

(18) `layers/affine_layer.m/backward`

(19) `layers/crossentropy_softmax_layer.m/forward`

(23) `layers/crossentropy_softmax_layer.m/forward`

(25) `layers/crossentropy_softmax_layer.m/backward`

(4) `layer/mlp.m/apply_gradient_descent_step`

2 Introduction

A ***multilayer perception*** (MLP) or ***fully-connected network*** (FCN), is a parametric mathematical function $\mathbf{f}(\mathbf{x};\boldsymbol{\theta}) : \mathcal{X} \to \mathcal{Y}$, where the parameters $\boldsymbol{\theta}$ are referred to as ***weights***. For shorthand, we'll drop writing $\boldsymbol{\theta}$ when it's obvious from context. The MLP has a special structure, which makes it efficient to implement, namely, it is has a modular layer-wise structure. Each layer $\mathbf{f}^\ell$ maps an input to an output. As such, the MLP can be written as the composition:

$$\mathbf{y}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) = \mathbf{f}^L(\mathbf{f}^{L-1}(\cdots \mathbf{f}^2(\mathbf{f}^1(\mathbf{x})))) \tag{1}$$

Each layer, in its simplest form, performs an affine transformation of its input $\mathbf{x}$ (row vector), followed by an ***element-wise nonlinearity*** σ^ℓ, so

$$\mathbf{f}^\ell(\mathbf{x}) = \sigma^\ell\left(\mathbf{x}\mathbf{W}^\ell + \mathbf{b}^\ell\right) = \sigma^\ell\left([\mathbf{x}, \mathbf{1}]\begin{bmatrix}\mathbf{W}^\ell \\ \mathbf{b}^\ell\end{bmatrix}\right) := \sigma^\ell\left(\tilde{\mathbf{x}}\tilde{\mathbf{W}}^\ell\right). \tag{2}$$

Here we have defined the 'augmented parameters' $\{\tilde{\mathbf{x}}, \tilde{\mathbf{W}}\}$. This trick should be known to you already. Typical element-wise nonlinearities are the once-popular sigmoid function or the now more common rectified linear unit (ReLU for short).

Maybe confusingly, when people make practical implementations of neural networks, they often break the traditional affine transformation + nonlinearity into two separate layers. So the affine transformation is a layer in itself and the nonlinearity is also a layer, but with no parameters.

3 Backpropagation

To train the MLP, we use the ***backpropagation algorithm***, an algorithmic implementation of the chain-rule of calculus. Given a dataset $\mathcal{D} = \{\mathbf{x}_n, \mathbf{t}_n\}_{n=1}^N$, with inputs $\mathbf{x}_n$ and targets $\mathbf{t}_n$, and a loss function $\mathcal{L}(\mathbf{t}_n, \mathbf{y}(\mathbf{x}_n; \boldsymbol{\theta}))$, we wish to set the parameters $\boldsymbol{\theta} = \{\tilde{\mathbf{W}}^\ell\}_{\ell=1}^L$, such that the loss function is minimized over the dataset. In maths, this is

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^N \mathcal{L}(\mathbf{t}_n, \mathbf{y}(\mathbf{x}_n; \boldsymbol{\theta})) \tag{3}$$

We perform this minimization using the ***stochastic gradient descent iteration***:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \lambda_t \left.\frac{\partial \bar{\mathcal{L}}}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}_t} \tag{4}$$

where λ_t is the learning rate, and $\bar{\mathcal{L}}$ is the averaged gradient over a ***minibatch*** of data points. Stochastic gradient descent differs from regular gradient descent in that stochastic GD uses a statistical estimate of the true loss gradient, based on a sample of the full training data. This is fast, but it has its issues and it is still a topic of open debate whether this is a sensible algorithm for optimisation.

To compute the gradients of the loss with respect to the parameters in layer ℓ, we use the chain rule adapted to matrix algebra.

Matrix Jacobians 101 Jacobians are the higher dimensional analogue of gradients. For an expression like $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$, we store the gradient for each element of $\mathbf{y}$ wrt $\mathbf{x}$ in a matrix like

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \cdots & \frac{\partial y_M}{\partial x_N} \end{bmatrix}. \tag{5}$$

So if y is a scalar, then $\frac{\partial y}{\partial \mathbf{x}}$ is a row vector and if x is a scalar, then $\frac{\partial \mathbf{y}}{\partial x}$ is a column vector. The chain rule works with this notation very smoothly. Convince yourself that if $\mathbf{y} = \mathbf{f}(\mathbf{x})$, where $\mathbf{f}$ is a vector-valued function, then

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{f}} \frac{\partial \mathbf{f}}{\partial \mathbf{x}}, \tag{6}$$

which is a simple matrix multiplication. When we have matrices on the top/bottom of one of these expressions, then things get a little more involved, but don't worry, you will never have to work with these quantities. In the rest of this cheatsheet, just pretend they are matrices. It works out in the end that all the equations you need to use are all matrix equations anyway.

The gradient of the loss function defined on a single training pair $\{\mathbf{x}, \mathbf{y}\}$, with respect to weight $\mathbf{W}_\ell$ in layer ℓ is

$$\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{W}}_\ell} = \frac{\partial \mathcal{L}}{\partial \mathbf{F}_\ell} \frac{\partial \mathbf{F}_\ell}{\partial \tilde{\mathbf{W}}_\ell} \tag{7}$$

We have written $\mathbf{F}_\ell$ as a shorthand for a minibatch of activations at layer ℓ, where each row of $\mathbf{F}_\ell$ corresponds to a separate training example.

Looking closer at the expression we have two different gradients. The layer-wise parameter gradients are for the form $\frac{\partial \mathbf{F}_\ell}{\partial \tilde{\mathbf{W}}_\ell}$. Although they are 3D arrays, we shall see that they are very easy to work with. The other term $\frac{\partial \mathcal{L}}{\partial \mathbf{F}_\ell}$ is the ***backpropagated error***. It is called back-propagated, because we can compute it in a recursive major, per-layer, working from the output ***back*** down the network. Specifically

$$\frac{\partial \mathcal{L}}{\partial \mathbf{F}_\ell} = \frac{\partial \mathcal{L}}{\partial \mathbf{F}_{\ell+1}} \frac{\partial \mathbf{F}_{\ell+1}}{\partial \mathbf{F}^\ell}. \tag{8}$$

What this equation says is that the back-propagated error at layer ℓ is the product of the back-propagated error at layer $\ell + 1$ and the gradient of the output of layer ℓ with respect to its input. Thus during back-propagation, we need to compute two quantities

1. $\frac{\partial \mathbf{F}_{\ell+1}}{\partial \mathbf{F}^\ell}$ so we can compute $\frac{\partial \mathcal{L}}{\partial \mathbf{F}_\ell} = \frac{\partial \mathcal{L}}{\partial \mathbf{F}_{\ell+1}} \frac{\partial \mathbf{F}_{\ell+1}}{\partial \mathbf{F}^\ell}$
2. $\frac{\partial \mathbf{F}_\ell}{\partial \tilde{\mathbf{W}}_\ell}$ so we can compute $\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{W}}_\ell} = \frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{F}}_\ell} \frac{\partial \mathbf{F}_\ell}{\partial \tilde{\mathbf{W}}_\ell}$.

Next we provide the gradients for each component in the exercise.

3.1 Gradients

CAVEAT: In this section we have two parallel notations. We use lower case bold letters (e.g., $\mathbf{x}$) for row vectors, we use upper case bold letters (e.g., for matrices), and we use unbolded letters (e.g., t_i or Z) for scalars.

For easy reading, we drop the layer index ℓ, we relabel the input to each layer as $\mathbf{X}$, and the output from each layer as $\mathbf{Y}$. We use $\bullet$ to mean element-wise multiplication, trace($\mathbf{X}$) $= \sum_i \mathbf{X}_{ii}$ and $\mathbf{1}$ is a row vector of ones. To make the maths more approachable, we provide intermediate working out for a single training datum and simply state final results for minibatches in blue. It's up to the interested student to derive the minibatch gradients.

3.1.1 ReLU

The Rectified Linear Unit (ReLU) is a nonlinearity

$$\mathbf{y} = \mathrm{ReLU}(\mathbf{x}) \tag{9}$$
$$\mathbf{Y} = \max\{\mathbf{0}, \mathbf{X}\} \tag{10}$$

It has no parameters, so we need not compute $\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{W}}_\ell}$. The gradient wrt the input is just

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathrm{diag}(\mathbb{I}(\mathbf{x} > 0)) \tag{11}$$

$\mathbb{I}$ is the element-wise indicator function, which returns a 1 if the statement in the brackets is true. The back-propagated error for a minibatch is (using $\bullet$ for element-wise multiplication)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \bullet \mathbb{I}[\mathbf{X} > 0] \tag{12}$$

For the interested student Technically the gradient does not exist at $x = 0$ and instead we should instead use something called a subgradient, which is above and beyond this course. Stephen Boyd at Stanford has a good course on these, which are online as of 13 Dec 2016, Course EE364b. In the end, the conditions for convergence are identical to the conditions for SGD. For the ReLU, the value of the subgradient can be anything in the interval $[0, 1]$, so we picked 0.

3.1.2 Affine layer

The affine layer is (***pay attention to the tilde***—note that in the code we write `obj.W` $= \tilde{\mathbf{W}}$)

$$\mathbf{y} = \tilde{\mathbf{x}}\tilde{\mathbf{W}} = \mathbf{b} + \mathbf{x}\mathbf{W} \tag{13}$$
$$\mathbf{Y} = \tilde{\mathbf{X}}\tilde{\mathbf{W}} = \mathbf{1}^\top \mathbf{b} + \mathbf{X}\mathbf{W} \tag{14}$$

This has gradients

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial}{\partial \mathbf{x}} \left(\mathbf{b}^\top + \mathbf{W}^\top \mathbf{x}^\top \right) = \mathbf{W}^\top \tag{15}$$
$$\frac{\partial \mathbf{y}}{\partial \tilde{\mathbf{W}}} = \frac{\partial}{\partial \tilde{\mathbf{W}}} \tilde{\mathbf{W}}^\top \tilde{\mathbf{x}}^\top = \tilde{\mathbf{x}} \tag{16}$$

The parameter gradients are the same for a minibatch as for a single datum in this case. Notice how we form the back-propagated errors using $\mathbf{x}$ and not $\tilde{\mathbf{x}}$. So when we compute the summed gradients for gradient descent

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Y}} \mathbf{W}^\top \tag{17}$$
$$\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{W}}} = \mathbf{X}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{Y}}. \tag{18}$$

3.1.3 Softmax layer

The softmax layer is

$$\mathbf{y} = \frac{1}{Z}\exp\{\mathbf{x}\} \qquad Z = \sum_i \exp\{\mathbf{x}_i\}. \tag{19}$$

We have used exp to mean the element-wise exponential function. This has no parameters so its gradient wrt input is (using the product rule)

$$\begin{aligned}\frac{\partial \mathbf{y}}{\partial \mathbf{x}} &= \frac{1}{Z}\frac{\partial \exp\{\mathbf{x}\}}{\partial \mathbf{x}} + \exp\{\mathbf{x}\}\frac{\partial Z^{-1}}{\partial \mathbf{x}} && (20)\\ &= \frac{1}{Z}\mathrm{diag}(\exp\{\mathbf{x}\}) - \frac{1}{Z^2}\exp\{\mathbf{x}\}\frac{\partial Z}{\partial \mathbf{x}} && (21)\\ &= \mathrm{diag}(\mathbf{y}) - \mathbf{y}\mathbf{y}^T && (22)\end{aligned}$$

We won't compute the back-propagated error for this layer, because in practice we will not use it (see next part).

3.1.4 Crossentropy layer

The cross-entropy layer is

$$\mathcal{L} = -\frac{1}{M}\sum_m\sum_i t_{mi}\log y_{mi} = -\frac{1}{M}\sum_m\sum_i(\mathbf{T}\bullet\log\mathbf{Y})_{mi} \tag{23}$$

where $\mathbf{T}$ is a binary ***target***, we have used log to mean an element-wise logarithm, m indexes over the M data points in a minibatch, and i indexes over the dimensions of the output. The gradient of this wrt $\mathbf{Y}$ is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Y}} = \frac{\partial}{\partial \mathbf{Y}}\mathrm{trace}((\log\mathbf{Y})^\top\mathbf{T}) = -\mathbf{T}./\mathbf{Y}, \tag{24}$$

where ./ means element-wise division. Note that the input to this layer is usually the output of a softmax, which produces some outputs that are usually very close to 0. This makes the gradients very large and potentially numerically unstable. Solved in the next section.

3.1.5 Crossentropy-softmax

The back-propagated gradient of a crossentropy and softmax layer combined is stable and very simple. It is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}}\frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = \mathrm{softmax}(\mathbf{X}) - \mathbf{T}. \tag{25}$$

Can you derive this? Hint: write everything out using index notation instead of using matrices.

3.1.6 A stable softmax

The softmax itself is susceptible to instability, when some elements of its input are moderately large, since the exponential makes of the items in the denominator extremely large. We sort this, using a remapping. Note that

$$y_j = \frac{e^{x_j}}{\sum_i e^{x_i}} = \frac{e^{x_j - b}}{\sum_i e^{x_i - b}} \tag{26}$$

for any number b, since

$$\frac{e^{x_j-b}}{\sum_i e^{x_i-b}} = \frac{e^{-b}e^{x_j}}{e^{-b}\sum_i e^{x_i}} = \frac{e^{x_j}}{\sum_i e^{x_i}}. \tag{27}$$

To prevent numerical overflow, we set b adaptively to $\max_i x_i$, so the stable softmax is

$$y_j = \frac{e^{x_j-\max_i x_i}}{\sum_i e^{x_i-\max_i x_i}}. \tag{28}$$

4 Deriving the Robbins-Monro conditions*

For the interested student, we derive the conditions on the learning rate, so that convergence to a local minimum is guaranteed for stochastic gradient descent. The loss function at time T is

$$\mathcal{L}_{\text{TOTAL}} = \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}(\mathbf{t}_n, \mathbf{y}(\mathbf{x}_n;\boldsymbol{\theta}_T)) = \mathbb{E}_{n\sim\text{Uniform}\{1,...,N\}}\left[\mathcal{L}(\mathbf{t}_n, \mathbf{y}(\mathbf{x}_n;\boldsymbol{\theta}_T))\right]. \tag{29}$$

If we rewrite each item in the expectation as an individual loss function $\mathcal{L}^n(\boldsymbol{\theta}_T) := \mathcal{L}(\mathbf{t}_n, \mathbf{y}(\mathbf{x}_n;\boldsymbol{\theta}_T))$ then

$$\mathcal{L}_{\text{TOTAL}} = \mathbb{E}_n\left[\mathcal{L}^n(\boldsymbol{\theta}_T)\right]. \tag{30}$$

When we run stochastic gradient descent (SGD), we can think of the gradient of the loss define on each training point then as a random gradient from a distribution over gradients

$$\frac{\partial\mathcal{L}^n}{\partial\boldsymbol{\theta}_T} \sim \text{Uniform}\left\{\frac{\partial\mathcal{L}^1}{\partial\boldsymbol{\theta}_T}, ..., \frac{\partial\mathcal{L}^N}{\partial\boldsymbol{\theta}_T}\right\}. \tag{31}$$

So an SGD step looks like

$$\boldsymbol{\theta}_{T+1} = \boldsymbol{\theta}_T - \lambda_T\frac{\partial\mathcal{L}^n}{\partial\boldsymbol{\theta}_T} := \boldsymbol{\theta}_T - \lambda_T\mathbf{g}_T^n. \tag{32}$$

λ_T is the learning rate at iteration T. Since the gradient is a random variable, thus $\boldsymbol{\theta}_{T+1}$ is a random variable, depending deterministically on the set of sampled training points (and their order), up until iteration T. If we assume that the loss function is convex, then there is a single $\boldsymbol{\theta}_*$, which minimizes $\mathcal{L}_{\text{TOTAL}}$. Next we bound the magnitude of the squared Euclidean distance of the current parameters from the minimizer and find the conditions on $\boldsymbol{\theta}$ to drive this squared distance down to zero in expectation.

$$\begin{aligned}
\mathbb{E}_n\|\boldsymbol{\theta}_{T+1} - \boldsymbol{\theta}_*\|_2^2 &= \mathbb{E}_n\left[\|\boldsymbol{\theta}_T - \lambda_T\mathbf{g}_T^n - \boldsymbol{\theta}_*\|_2^2\right] &(33)\\
&= \mathbb{E}_n\left[\|\boldsymbol{\theta}_T - \boldsymbol{\theta}_*\|_2^2 - 2\lambda_T(\boldsymbol{\theta}_T - \boldsymbol{\theta}_*)^\top\mathbf{g}_T^n + \lambda_T^2\|\mathbf{g}_T^n\|_2^2\right] &(34)\\
&= \mathbb{E}_n\left[\|\boldsymbol{\theta}_T - \boldsymbol{\theta}_*\|_2^2 - 2\lambda_T(\boldsymbol{\theta}_T - \boldsymbol{\theta}_*)^\top\mathbf{g}_T^n + \lambda_T^2\|\mathbf{g}_T^n\|_2^2\right] &(35)
\end{aligned}$$

Now since $\mathcal{L}_{\text{TOTAL}}$ is convex we can write the inequality

$$\mathcal{L}_{\text{TOTAL},*} \leq \mathcal{L}_{\text{TOTAL}}(\boldsymbol{\theta}_T) + (\boldsymbol{\theta}_* - \boldsymbol{\theta}_T)^\top \mathbf{g}_{\text{TOTAL},T} \tag{36}$$

$$\mathbb{E}_n \mathcal{L}_{\text{TOTAL},*} \leq \mathbb{E}_n \mathcal{L}_{\text{TOTAL}}(\boldsymbol{\theta}_T) + \mathbb{E}_n (\boldsymbol{\theta}_* - \boldsymbol{\theta}_T)^\top \mathbf{g}_{\text{TOTAL},T} \tag{37}$$

$$\mathbb{E}_n \mathcal{L}_*^n \leq \mathbb{E}_n \mathcal{L}^n(\boldsymbol{\theta}_T) + \mathbb{E}_n (\boldsymbol{\theta}_* - \boldsymbol{\theta}_T)^\top \mathbf{g}_T^n \tag{38}$$

$$\mathbb{E}_n \left[\mathcal{L}^n(\boldsymbol{\theta}_T) - \mathcal{L}_*^n\right] \leq \mathbb{E}_n \left[(\boldsymbol{\theta}_T - \boldsymbol{\theta}_*)^\top \mathbf{g}_T^n\right] \tag{39}$$

So returning to the error bound

$$\mathbb{E}_n \|\boldsymbol{\theta}_{T+1} - \boldsymbol{\theta}_*\|_2^2 = \mathbb{E}_n \left[\|\boldsymbol{\theta}_T - \boldsymbol{\theta}_*\|_2^2 - 2\lambda_T (\boldsymbol{\theta}_T - \boldsymbol{\theta}_*)^\top \mathbf{g}_T^n + \lambda_T^2 \|\mathbf{g}_T^n\|_2^2\right] \tag{40}$$

$$\leq \mathbb{E}_n \left[\|\boldsymbol{\theta}_T - \boldsymbol{\theta}_*\|_2^2 - 2\lambda_T \left[\mathcal{L}^n(\boldsymbol{\theta}_T) - \mathcal{L}_*^n\right] + \lambda_T^2 \|\mathbf{g}_T^n\|_2^2\right]. \tag{41}$$

Now we note that $0 < \mathbb{E}_n \|\boldsymbol{\theta}_{T+1} - \boldsymbol{\theta}_*\|_2^2$, and expand the term $\|\boldsymbol{\theta}_T - \boldsymbol{\theta}_*\|_2^2$ recursively.

$$0 \leq \mathbb{E}_n \left[\|\boldsymbol{\theta}_T - \boldsymbol{\theta}_*\|_2^2 - 2\lambda_T \left[\mathcal{L}^n(\boldsymbol{\theta}_T) - \mathcal{L}_*^n\right] + \lambda_T^2 \|\mathbf{g}_T^n\|_2^2\right] \tag{42}$$

$$= \mathbb{E}_n \left[\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_*\|_2^2 - 2\sum_{t=1}^{T} \lambda_t \left[\mathcal{L}^n(\boldsymbol{\theta}_t) - \mathcal{L}_*^n\right] + \sum_{t=1}^{T} \lambda_t^2 \|\mathbf{g}_t^n\|_2^2\right]. \tag{43}$$

Now we make a couple of assumptions. Assume

1. we start SGD within some region of convergence, of radius R so $\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_*\|_2^2 < R^2$
2. the gradient in this region never exceeds $\|\mathbf{g}_t^n\| < G^2$

This leads to

$$\mathbb{E}_n \left[2\sum_{t=1}^{T} \lambda_t \left[\mathcal{L}^n(\boldsymbol{\theta}_t) - \mathcal{L}_*^n\right]\right] \leq R^2 + G^2 \sum_{t=1}^{T} \lambda_t^2. \tag{44}$$

Now we make note of the inequality

$$\min_t \mathbb{E}_n \left[\mathcal{L}^n(\boldsymbol{\theta}_t) - \mathcal{L}_*^n\right] \cdot 2\sum_{t=1}^{T} \lambda_t \leq \mathbb{E}_n \left[2\sum_{t=1}^{T} \lambda_t \left[\mathcal{L}^n(\boldsymbol{\theta}_t) - \mathcal{L}_*^n\right]\right] \tag{45}$$

So the bound is now

$$\min_t \mathbb{E}_n \left[\mathcal{L}^n(\boldsymbol{\theta}_t) - \mathcal{L}_*^n\right] \cdot 2\sum_{t=1}^{T} \lambda_t \leq R^2 + G^2 \sum_{t=1}^{T} \lambda_t^2 \tag{46}$$

$$\min_t \mathbb{E}_n \left[\mathcal{L}^n(\boldsymbol{\theta}_t) - \mathcal{L}_*^n\right] \leq \frac{R^2 + G^2 \sum_{t=1}^{T} \lambda_t^2}{2\sum_{t=1}^{T} \lambda_t} \tag{47}$$

We can drive the minimum of the expected error to zero as $t \to \infty$, by setting $\sum_{t=1}^{\infty} \lambda_t = \infty$ and $\sum_{t=1}^{\infty} \lambda_t^2 < \infty$. These are the Robbins-Monro conditions.