On the Optimization of Neural Networks

Soufiane Belharbi

soufiane.belharbi@insa-rouen.fr

LITIS lab., Apprentissage team - INSA de Rouen, France
ASI 4

March, 22nd, 2017

Plan

Results

When you finish this course you will:

1. Understand the intuition behind the second order methods for optimization.
2. Understand Dropout (regularization). (today's bonus)

By doing the **Hands on Lab 2**, you will be able to:

1. Use Dropout to train a neural network using Keras.
2. Implement Dropout to train a neural network using Theano.

Function Approximation: Taylor Expansion

f is unknown function. $f(a)$ known for some values a.
$f'(a) = f^1(a)$ is known. The same for $f''(a) = f^2(a)$,
$f'''(a) = f^3(a), \ldots, f^n(a)$.
f can be approximated at any point x using a polynomial:

$$P(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \ldots$$

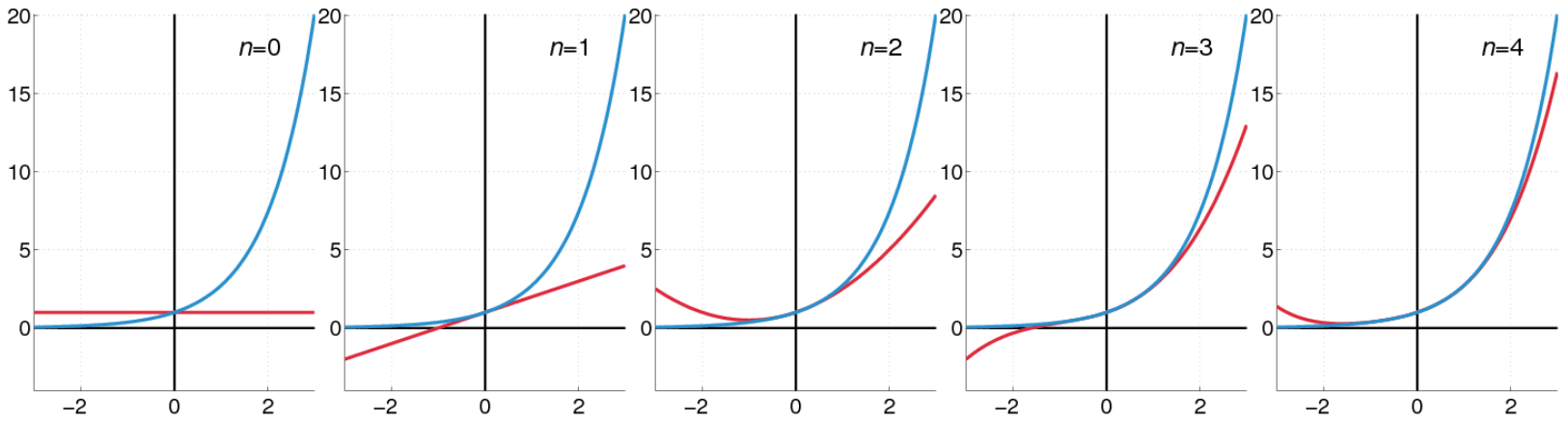


Figure 1: Approximation of exp^x (blue) using a polynomial of degree 4 (red)

Function Approximation: Taylor Expansion

Terms of the expansion:

$$P(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots$$

- $f(a)$: constant. (very bad approximation)
- $f'(a)(x-a)$: line (still bad)
- $\frac{f''(a)}{2!}(x-a)^2$: parabolic (curvature) (interesting)
- ...

Function Approximation: Taylor Expansion

Local approximation (variation h): $f(x) \simeq P(x)$

$P(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \ldots$

$P(a) = f(a)$, what happens when we variate a little bit x round a by small perturbation h?

$$\begin{aligned} P(x = a+h) &= f(a) + \frac{f'(a)}{1!}(a+h-a) + \frac{f''(a)}{2!}(a+h-a)^2 + \\ &+ \frac{f'''(a)}{3!}(a+h-a)^3 + \ldots \\ &= f(a) + \frac{f'(a)}{1!}h + \frac{f''(a)}{2!}h^2 + \frac{f'''(a)}{3!}h^3 + \ldots \end{aligned}$$

We are interested in 1^{st} and 2^{nd} order approximation.

$f(x) \simeq f(a) + f'(a)h$ (linear approximation)

and

$f(x) \simeq f(a) + f'(a)h + \frac{1}{2}f''(a)h^2$ (quadratic approximation)

Consider a loss function $\boldsymbol{E}(\mathbf{y}, f(\mathbf{x}; \mathbf{w}))$. $f(\mathbf{x}; \mathbf{w})$ is a function (network output) with parameters $\mathbf{w}$. For short, we refer to the error by $\boldsymbol{E}(\mathbf{w})$.
Consider the iterative update rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\partial \boldsymbol{E}(\mathbf{w})}{\partial \mathbf{w}}$$

How to approximate the derivative of the error with respect to the parameters?

First Order

$\boldsymbol{E}$ can be approximated locally at some point $\hat{\mathbf{w}}$ using first order information (linearly):

$\boldsymbol{E}(\mathbf{w}) \simeq \boldsymbol{E}(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})\frac{\partial E(\hat{\mathbf{w}})}{\partial \mathbf{w}} = \boldsymbol{E}(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})\nabla \boldsymbol{E}\big|_{\hat{\mathbf{w}}}$

Therefore, $\nabla \boldsymbol{E} \simeq \nabla \boldsymbol{E}\big|_{\mathbf{w}=\hat{\mathbf{w}}}$. (gradient approximated with a constant)

This is the Gradient Descent method.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla \boldsymbol{E} \Leftrightarrow \Delta \mathbf{w}_t = -\eta \nabla \boldsymbol{E}$$

Requires many steps to reach the minimum. Requires small batch sizes.

Second Order

$\boldsymbol{E}$ can be approximated locally at some point $\hat{\mathbf{w}}$ using first and second order information (quadratic):

$$\begin{aligned}\boldsymbol{E}(\mathbf{w}) &\simeq \boldsymbol{E}(\hat{\mathbf{w}}) + (\mathbf{w}-\hat{\mathbf{w}})\frac{\partial E(\hat{\mathbf{w}})}{\partial \mathbf{w}} + \frac{1}{2}(\mathbf{w}-\hat{\mathbf{w}})^2\frac{\partial^2 E(\hat{\mathbf{w}})}{\partial \mathbf{w}^2} \\ &= \boldsymbol{E}(\hat{\mathbf{w}}) + (\mathbf{w}-\hat{\mathbf{w}})\nabla\boldsymbol{E}\big|_{\hat{\mathbf{w}}} + \frac{1}{2}(\mathbf{w}-\hat{\mathbf{w}})^2\nabla\nabla\boldsymbol{E}\big|_{\hat{\mathbf{w}}}\end{aligned}$$

Therefore, $\nabla\boldsymbol{E} \simeq \nabla\boldsymbol{E}\big|_{\mathbf{w}=\hat{\mathbf{w}}} + (\mathbf{w}-\hat{\mathbf{w}})\nabla\nabla\boldsymbol{E}\big|_{\mathbf{w}=\hat{\mathbf{w}}}$. Where: $\nabla\nabla\boldsymbol{E}$ is the **Hessian matrix** $\mathbb{H}$ with elements: $(\mathbb{H})_{ij} = \frac{\partial \boldsymbol{E}}{\partial \mathbf{w}_i \partial \mathbf{w}_j}\Big|_{\mathbf{w}=\hat{\mathbf{w}}}$. The gradient is approximated with a line: $\nabla\boldsymbol{E} \simeq b + \mathbb{H}(\mathbf{w}-\hat{\mathbf{w}})$ with: $b = \nabla\boldsymbol{E}\big|_{\hat{\mathbf{w}}}$.

Note that at the optimum $\mathbf{W}_{min}$ (i.e. $\frac{\partial \boldsymbol{E}}{\partial \mathbf{w}}\Big|_{\mathbf{w}=\mathbf{w}_{min}} = 0$), starting from $\hat{\mathbf{w}}$, we can reach $\mathbf{W}_{min}$ in one step:

$$\begin{aligned}\frac{\partial \boldsymbol{E}}{\partial \mathbf{w}}\Big|_{\mathbf{w}=\mathbf{w}_{min}} = 0 &\Leftrightarrow \nabla\boldsymbol{E}\Big|_{\mathbf{w}=\hat{\mathbf{w}}} + (\mathbf{w}_{min}-\hat{\mathbf{w}})\mathbb{H}\Big|_{\mathbf{w}=\hat{\mathbf{w}}} \\ &\Leftrightarrow \mathbf{w}_{min} = \hat{\mathbf{w}} - (\mathbb{H}\Big|_{\mathbf{w}=\hat{\mathbf{w}}})^{-1}\nabla\boldsymbol{E}\Big|_{\mathbf{w}=\hat{\mathbf{w}}} \\ &\Leftrightarrow \Delta\mathbf{w}_{\hat{\mathbf{w}}\rightarrow\mathbf{w}_{min}} = (\mathbb{H}\Big|_{\mathbf{w}=\hat{\mathbf{w}}})^{-1}\nabla\boldsymbol{E}\Big|_{\mathbf{w}=\hat{\mathbf{w}}}\end{aligned}$$

This is the second order base-method.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbb{H}^{-1}\nabla\boldsymbol{E}$$

If N is the total size of parameters of the network, $\mathbb{H}$ will be a squared matrix with size $N \times N$. $\mathbb{H}$ measures the curvature of $\boldsymbol{E}$. Requires larger batch sizes (reduces fluctuation in estimating $\mathbb{H}^{-1}\nabla\boldsymbol{E}$). Condition number issue.

Variations of Second Order Methods

- Newton algorithm: invert $\mathbb{H}$ + works only in batch mode.
- Conjugate gradient: does not use explicitly $\mathbb{H}$ but requires line search + works only in batch mode.
- Quasi-Newton (BFGS): approximate $\mathbb{H}^{-1}$ + line search + works only in batch mode.
- Gauss-Newton: approximate $\mathbb{H}$ using the square Jacobian. Used batch mode + works only for mean-squared error loss functions.
- Levenberg Marquardt method: Extends Gauss-Newton to include regularization parameter.

Pseudo-code for Implementing Newton 's Method

Algorithm 1 Newton's method with loss function: $\frac{1}{m}\sum_{i=1}^{m}\mathcal{C}(y_i, f(x_i; \mathbf{w}))$

1: $\mathcal{D}$ training set. $\mathbf{w}_0$ initial guess.
2: **while** stopping criterion not met **do**
3: Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\mathbf{w}}\sum_i \mathcal{C}(y_i, f(x_i; \mathbf{w}))$
4: Compute Hessian: $\mathbb{H} \leftarrow \frac{1}{m}\nabla^2_{\mathbf{w}}\sum_i \mathcal{C}(y_i, f(x_i; \mathbf{w}))$
5: Compute Hessian inverse: $\mathbb{H}^{-1}$
6: Compute update: $\Delta\mathbf{w} = -\mathbb{H}^{-1}\boldsymbol{g}$
7: Apply update: $\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$
8: **end while**

On the Hessian $\mathbb{H}$

- $\mathbb{H}$ is more appropriate (for example in Newton's method) when $\mathbb{H}$ is positive definite.
- $\mathbb{H}$ is positive definite $\Leftrightarrow \forall z \in \mathcal{R}^N : z^T \mathbb{H} z > 0 \Leftrightarrow \forall \lambda_i, \lambda_i > 0, \lambda_i$ eigenvalue of $\mathbb{H}$.
- Saddle points are a nightmare.
- If not all the eigenvalues are positive near saddle points, Newton's method can go in the wrong direction: regularized Hessian: $\mathbb{H}_{reg} = \mathbb{H} + \alpha I$. (Levenberg–Marquardt algorithm)
- The eigenvectors (directions) length is inversely proportional to the square root of the corresponding eigenvalues = the curvature is extremely steep toward small axis and very flat along the long axis. (taco-shell shaped minima)
- Typically in $\{\lambda_i\}$: there is small, medium and large values. **The large eigenvalues** will cause trouble during the training process (condition number of $\mathbb{H}$).

Tricks for the Second Order

Replace the **exact** $\mathbb{H}$ with an **approximation** of either the full or partial $\mathbb{H}$.

- Finite Differences (more than 2 back-propagations to estimate $\mathbb{H}$).
- Square Jacobian.
- Just compute the **diagonal** of $\mathbb{H}$.

All done using back-propagation.

For very large networks, $\mathbb{H}$ is very expensive and the optimization is very slow (because the batch mode). Therefore, on-line training is needed.

In many methods, all what we need is to calculate **the product of $\mathbb{H}$ with an arbitrary vector**. Luckily, this can be done without estimating $\mathbb{H}$.

Different techniques to **compute the principal eigenvalue and vector** without having to compute the $\mathbb{H}$: the power method, Taylor expansion, on-line method.

Neural Networks and $\mathbb{H}$

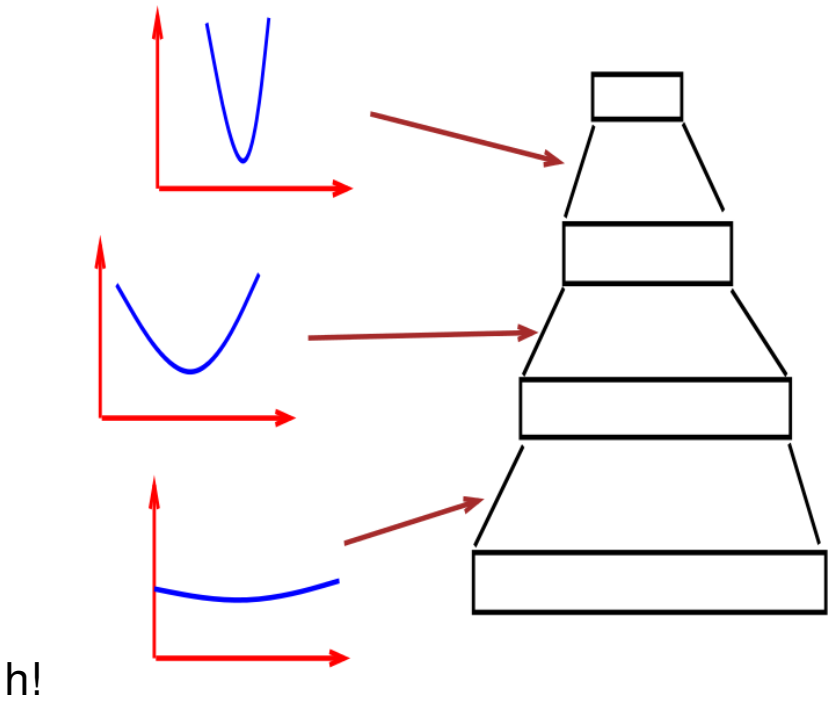
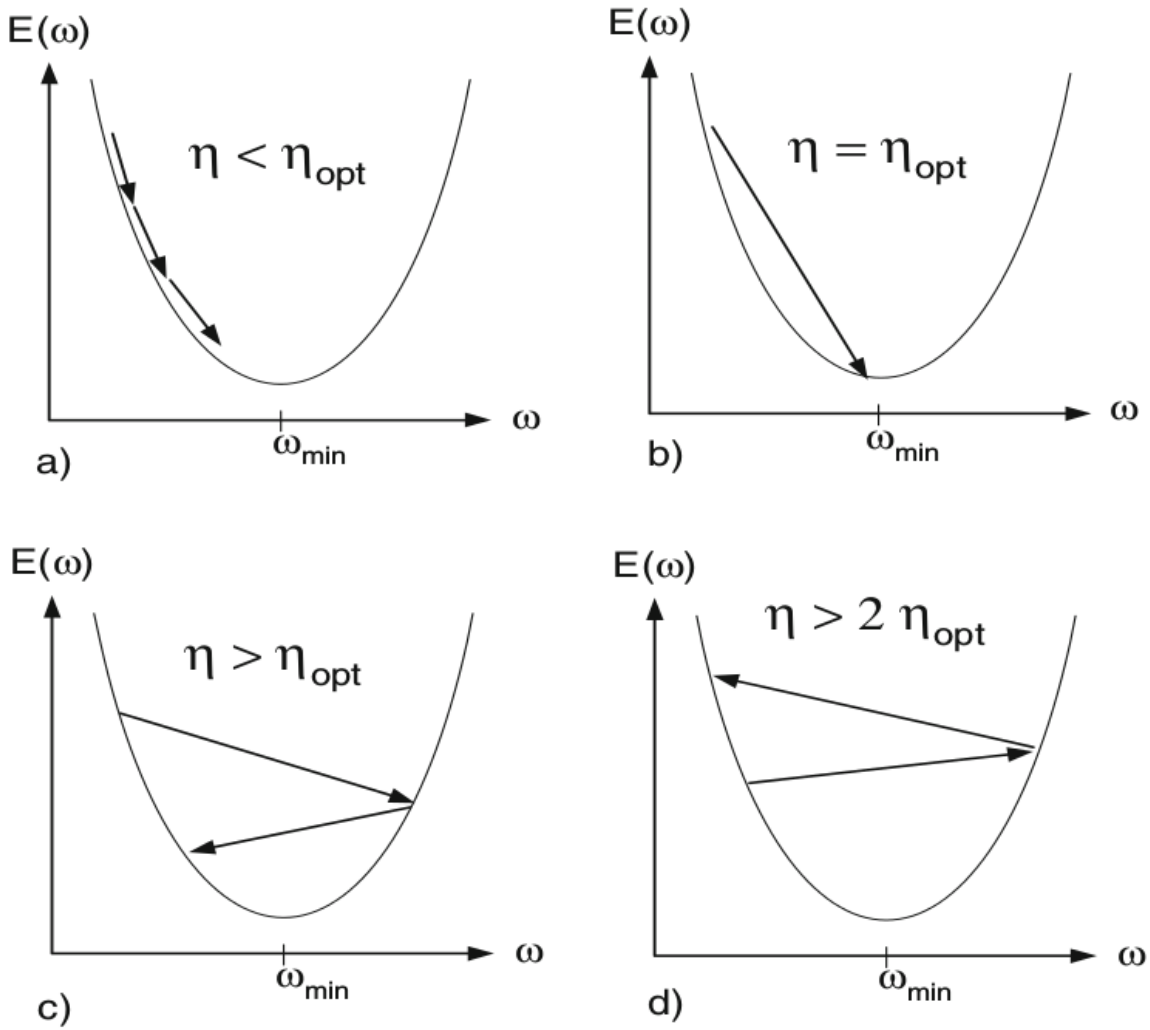


Figure 2: (Blue): second derivative: shape of the Hessian $\mathbb{H}$.

- Learning speed: lower layers going slow, last layer going faster (may oscillate).
- Small changes in the input may lead to large changes in the output. (deeper networks)

On the Importance of the Learning Rate



To see in more details in future courses.

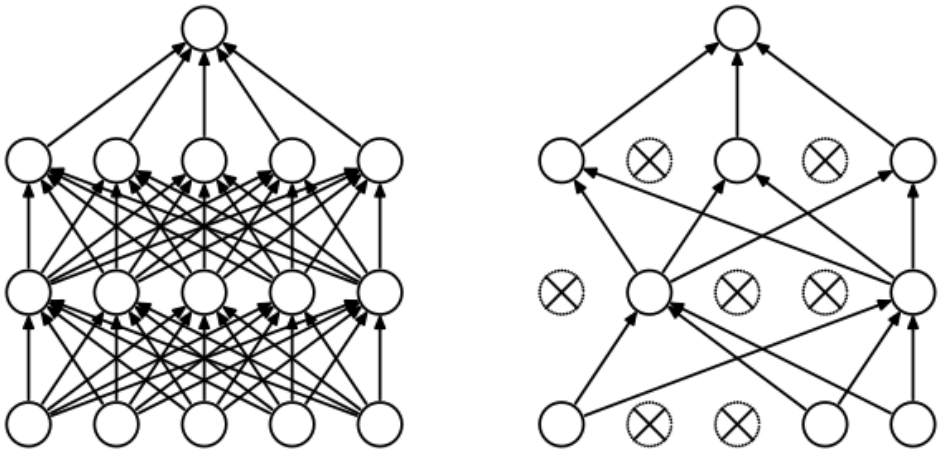
On the Importance of the Learning Rate

- The eigenvalues of $\mathbb{H}$ measure the steepness of $\boldsymbol{E}$ along the corresponding eigendirections.
- Using single learning rate for the whole network parameters is problematic: we want η to be large along shallow direction of $\boldsymbol{E}$ (small eigenvalues of $\mathbb{H}$) and to be small in the steepest directions of $\boldsymbol{E}$ (eigenvalues of $\mathbb{H}$ is large). Assign a learning rate to each direction?
- If we want to use a single learning rate η, then in order to avoid divergence: $\eta < \frac{2}{\lambda_{max}}$, where λ_{max} is the maximum eigenvalue of $\mathbb{H}$.

- The second order methods ($\mathbb{H}$) are interesting tools to study the curvature of the error surface. But, they are very expensive (storage, computation).
- Second order methods are impractical for training neural networks. Stick with first order and try to improve it based on insights from 2^{nd} order methods.

Dropout: a Simple Way to Prevent Over-fitting

What is it? training technique to reduce the over-fitting a neural network.
How to do it? during the training, **turn off some random nodes** when forwarding.

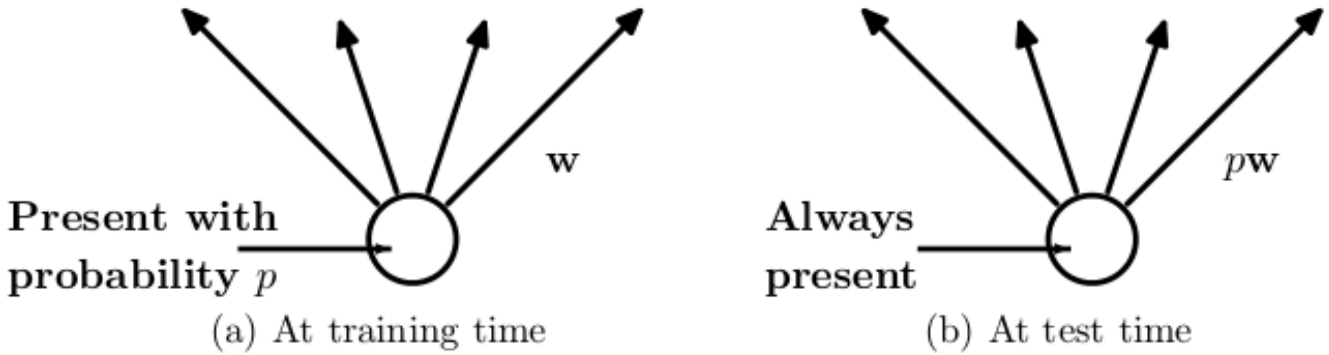


(a) Standard Neural Net (b) After applying dropout.

Figure 3: Crossed units were dropped.

Dropout: a Simple Way to Prevent Over-fitting

How to do it?
For a given layer: **drop** d% of neurones. $\Leftrightarrow$ probability of a neurone to be present is: $p = 1 - d$%. Example: in a layer with size: 120. Drop 70 neurones means: $d = \frac{70}{120} \simeq 0.5833$. $p \simeq 1 - 0.5833 \simeq 0.4122$.



(a) At training time (b) At test time

h!

For a layer of size s:

1. Train: for **each forward, generate a binary mask** using a binomial $B(n = 1, p = p)$ distribution with size = s. Multiply the input layer by this mask.
2. Test: When building the network, multiply each weight by p. Nothing else to do.

Dropout: a Simple Way to Prevent Over-fitting

How to do it? (more general)
Consider a network with L layers. $l \in \{1, \ldots, L\}$. $z^{(l)}$ is the input of the layer l. $z^{(0)} = x$. $W^{(l)}$ and $b^{(l)}$ are the weights and the biases of the layer l. $\varphi^{(l)}(.)$ is the activation of the layer l.
Standard feed-forward:

$$z^{(l+1)} = w_i^{(l+1)} y^{(l)} + b_i^{(l+1)},$$
$$y_i^{(l+1)} = \varphi^{(l)}(z^{(l+1)}),$$

Feed-forward with dropout:

$$r_i^l \sim Bernoulli(p),$$
$$\hat{y}^{(l)} = r^{(l)} * y^{(l)},$$
$$z^{(l+1)} = w_i^{(l+1)} \hat{y}^{(l)} + b_i^{(l+1)}$$
$$y_i^{(l+1)} = \varphi^{(l)}(z^{(l+1)})$$

p **can be different at each layer**.

Dropout: a Simple Way to Prevent Over-fitting

How to do it? (**Practice**)
In practice: you can build a **dropout layer** which has only **binomial random generator**. This layer does not have any parameters. It has an input and an output and perform the dropout as follows:

1. Take the input.
2. Generate the random mask.
3. Multiply the input by the mask.
4. The result of the multiplication is the output.

The dropout reduces the over-fitting of the network. Usually, it is applied for fully connected layers. But, it can be applied for convolutional layers as well.
How to determine the best p (s)? use validation. (typical values: $20, 30, 40, 50\%$).
Using dropout over the input data is equivalent to using noise.

Questions

Thank you for your attention,

Questions?

Is that all?
Yes.
What now?
Hands on Lab 2.

Hands on Lab 2

Break for 15 minutes.

. . . .

1. Use Dropout to train a neural network using Keras.
2. Implement Dropout to train a neural network using Theano.