

REGULARIZATION IN NEURAL NETWORKS

MASTER'S DEEP LEARNING

Sergey Nikolenko

Harbour Space University, Barcelona, Spain

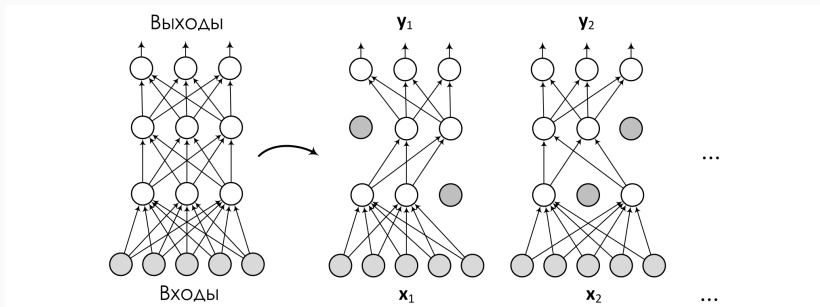
November 13, 2017

REGULARIZATION IN NEURAL NETWORKS

- NNs have *lots* of parameters.
- Regularization is necessary.
- L_2 or L_1 regularization ($\lambda \sum_w w^2$ or $\lambda \sum_w |w|$) is called *weight decay*.
- Very easy to add, just another term in the objective function.
- Sometimes still useful.

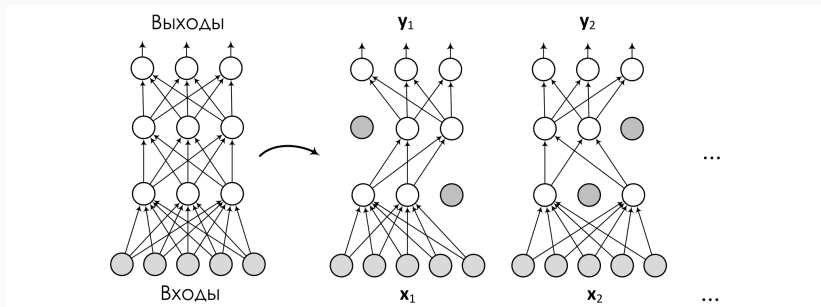
REGULARIZATION IN NEURAL NETWORKS

- But there are better ways.
- *Dropout*: remove some units at random with probability p !



REGULARIZATION IN NEURAL NETWORKS

- To apply, simply multiply the result by $1/p$ (preserving average output); and you can usually take $p = \frac{1}{2}$.



- Dropout improved everything *drastically*. What the... why does it work?
- Idea 1: we are making the units learn features by themselves, without relying on the others.
- Idea 2: we are kind of *averaging* a huge number of networks with shared weights, training each for one step. Like bootstrapping taken to the extreme.
- Idea 3: this is just like sex!
- Idea 4: dropout is a special kind of prior (this has led to proper dropout in recurrent NNs).

WEIGHT INITIALIZATION

- The deep learning revolution began with *unsupervised pretraining*.
- Main idea: get to a good region of the search space, then fine-tune with gradient descent.
- Turns out by now we don't need unsupervised pretraining with complex models like RBM to get to a good region.
- Weight initialization is an important part of why.

- *Xavier initialization* (Glorot, Bengio, 2010).
- Let's consider a single linear unit:

$$y = \mathbf{w}^\top \mathbf{x} + b = \sum_i w_i x_i + b.$$

- The variance is

$$\begin{aligned} \text{Var}[y_i] &= \text{Var}[w_i x_i] = \mathbb{E}[X^2 Y^2] - (\mathbb{E}[XY])^2 = \\ &= \mathbb{E}[x_i]^2 \text{Var}[w_i] + \mathbb{E}[w_i]^2 \text{Var}[x_i] + \text{Var}[w_i] \text{Var}[x_i]. \end{aligned}$$

- The variance is

$$\begin{aligned}\text{Var}[y_i] &= \text{Var}[w_i x_i] = \mathbb{E}[X^2 Y^2] - (\mathbb{E}[XY])^2 = \\ &= \mathbb{E}[x_i]^2 \text{Var}[w_i] + \mathbb{E}[w_i]^2 \text{Var}[x_i] + \text{Var}[w_i] \text{Var}[x_i].\end{aligned}$$

- For symmetric activation functions and zero mean of the weights

$$\text{Var}[y_i] = \text{Var}[w_i] \text{Var}[x_i].$$

- And if w_i and x_i are initialized independently from the same distribution,

$$\text{Var}[y] = \text{Var}\left[\sum_{i=1}^{n_{\text{out}}} y_i\right] = \sum_{i=1}^{n_{\text{out}}} \text{Var}[w_i x_i] = n_{\text{out}} \text{Var}[w_i] \text{Var}[x_i].$$

- In other words, the output variance is proportional to the input variance with coefficient $n_{\text{out}} \text{Var}[w_i]$.

- Before (Glorot, Bengio, 2010), the standard way to initialize was (it's all over older literature)

$$w_i \sim U \left[-\frac{1}{\sqrt{n_{\text{out}}}}, \frac{1}{\sqrt{n_{\text{out}}}} \right].$$

- So in this case we get

$$\text{Var}[w_i] = \frac{1}{12} \left(\frac{1}{\sqrt{n_{\text{out}}}} + \frac{1}{\sqrt{n_{\text{out}}}} \right)^2 = \frac{1}{3n_{\text{out}}}, \text{ so}$$

$$n_{\text{out}} \text{Var}[w_i] = \frac{1}{3},$$

and after a few layers the signal dies down; the same happens in backprop.

- Xavier initialization tries to reduce the change in variance, so we take

$$\text{Var}[w_i] = \frac{2}{n_{\text{in}} + n_{\text{out}}},$$

which for uniform distribution means

$$w_i \sim U \left[-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}} \right].$$

- But it only works for symmetric activations, i.e., not for ReLU...

- ...until (He et al., 2015)! Let's go back to

$$\text{Var} [w_i x_i] = \mathbb{E} [x_i]^2 \text{Var} [w_i] + \mathbb{E} [w_i]^2 \text{Var} [x_i] + \text{Var} [w_i] \text{Var} [x_i]$$

- We now can only make the second term zero:

$$\text{Var} [w_i x_i] = \mathbb{E} [x_i]^2 \text{Var} [w_i] + \text{Var} [w_i] \text{Var} [x_i] = \text{Var} [w_i] \mathbb{E} [x_i^2], \text{ so}$$

$$\text{Var} [y^{(l)}] = n_{\text{in}}^{(l)} \text{Var} [w^{(l)}] \mathbb{E} [(x^{(l)})^2].$$

- We now can only make the second term zero:

$$\text{Var} [y^{(l)}] = n_{\text{in}}^{(l)} \text{Var} [w^{(l)}] \mathbb{E} [(x^{(l)})^2].$$

- Suppose now that $x^{(l)} = \max(0, y^{(l-1)})$, and $y^{(l-1)}$ has a symmetric distribution around zero. Then

$$\mathbb{E} [(x^{(l)})^2] = \frac{1}{2} \text{Var} [y^{(l-1)}], \quad \text{Var} [y^{(l)}] = \frac{n_{\text{in}}^{(l)}}{2} \text{Var} [w^{(l)}] \text{Var} [y^{(l-1)}].$$

- And this leads to the variance for ReLU init; there is no n_{out} now:

$$\text{Var} [w_i] = 2/n_{\text{in}}^{(l)}.$$

- You don't have to make it uniform, btw; e.g., a normal distribution is fine:

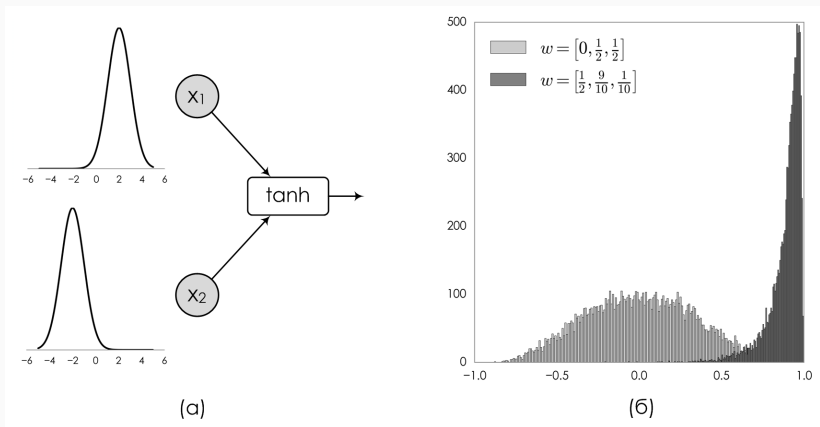
$$w_i \sim \mathcal{N} \left(0, \sqrt{2/n_{\text{in}}^{(l)}} \right).$$

BATCH NORMALIZATION

- Important problem in deep neural networks: *internal covariate shift*.
- When we change the weights of a layer, the distribution of its outputs changes.
- This means that the next layer has to re-train almost from scratch, it did not expect these outputs.
- Moreover, these neurons might have already reached saturation, so they can't re-train quickly.
- This seriously impedes learning.

BATCH NORMALIZATION

- A characteristic example; note how different the distributions are:



- What can we do?

- We could try to normalize (whiten) after every layer.
- Does not work: consider a layer that simply adds a bias b to its inputs u :

$$\hat{\mathbf{x}} = \mathbf{x} - \mathbb{E}[\mathbf{x}], \text{ where } \mathbf{x} = u + b.$$

- On the next gradient descent step, we'll have $b := b + \Delta b$...
- ...but $\hat{\mathbf{x}}$ will not change:

$$u + b + \Delta b - \mathbb{E}[u + b + \Delta b] = u + b - \mathbb{E}[u + b].$$

- So the biases will simply increase unboundedly, and that's all the training we'll get; not a good thing.

- We can try to add normalization as a layer:

$$\hat{\mathbf{x}} = \text{Norm}(\mathbf{x}, \mathcal{X}).$$

- But note that the entire dataset \mathcal{X} is required here.
- So on the gradient descent step we'll need to compute $\frac{\partial \text{Norm}}{\partial \mathbf{x}}$ and $\frac{\partial \text{Norm}}{\partial \mathcal{X}}$, and also the covariance matrix

$$\text{Cov}[\mathbf{x}] = \mathbb{E}_{\mathbf{x} \in \mathcal{X}} [\mathbf{x}\mathbf{x}^\top] - \mathbb{E}[\mathbf{x}] \mathbb{E}[\mathbf{x}]^\top.$$

- Definitely won't work.

- The solution is to normalize each component separately, and not over the whole dataset but over the current mini-batch; hence *batch normalization*.
- After batch normalization we get

$$\hat{x}_k = \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\text{Var}[x_k]}}$$

where the statistics are computed over the current mini-batch.

- However, one more problem: now nonlinearities disappear!
- E.g., we will almost always get into the region where σ is very close to linear.

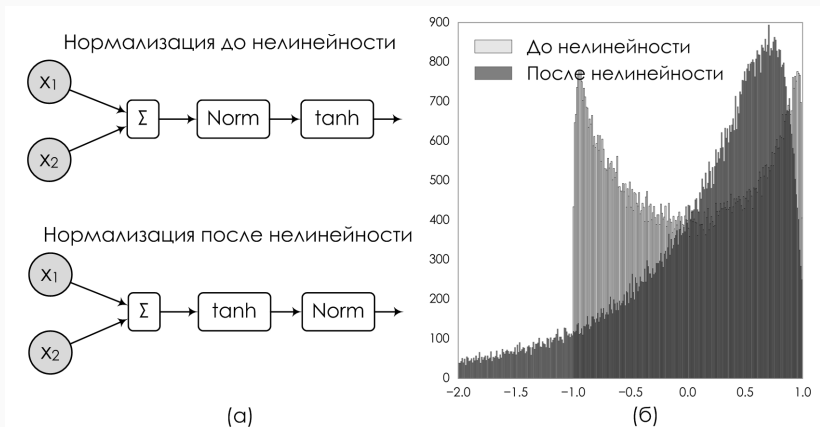
- To fix this, we have to allow the batchnorm layer enough flexibility to sometimes do *nothing* with the inputs.
- So we introduce additional shift and scale parameters:

$$y_k = \gamma_k \hat{x}_k + \beta_k = \gamma_k \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\text{Var}[x_k]}} + \beta_k.$$

- γ_k and β_k are new variables and will be trained just like the weights.

BATCH NORMALIZATION

- Last remark: it matters where to put the batchnorm.
- You can put it either before or after the nonlinearity.



VARIATIONS OF GRADIENT DESCENT

- Gradient descent:

$$\theta_t = \theta_{t-1} - \eta \nabla E(\mathbf{x}_t, \theta_{t-1}, y_t).$$

- It all depends on the learning rate η .
- First idea – let's make it decrease over time:

- linear decay:

$$\eta = \eta_0 \left(1 - \frac{t}{T}\right);$$

- exponential decay:

$$\eta = \eta_0 e^{-\frac{t}{T}}.$$

- But this does not take E into account; it's better to be *adaptive*.

- *Momentum methods*: let's keep part of the speed, like a real material point would.
- With the inertia we now have

$$u_t = \gamma u_{t-1} + \eta \nabla_{\theta} E(\theta),$$
$$\theta = \theta - u_t.$$

- So we now preserve γu_{t-1} .

- But we already know we will go to γu_{t-1} !
- Why don't we compute the gradients right there, halfway?
- *Nesterov's momentum*:

$$u_t = \gamma u_{t-1} + \eta \nabla_{\theta} E(\theta - \gamma u_{t-1})$$

- Can we do even better?..

- ...well, yeah, we can try second-order methods.
- Newton's method:

$$E(\theta) \approx E(\theta_0) + \nabla_{\theta} E(\theta_0)(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^{\top} H(E(\theta))(\theta - \theta_0).$$

- This is usually much faster, and there's nothing to tune (no η).
- But we need to compute the Hessian $H(E(\theta))$, and this is infeasible.
- Interesting problem: can we make Newton's method work for deep learning?

- But we can still do better!
- Note that so far the learning rate was the same in all directions.
- Idea: rate of change should be higher for parameters that do not change much over the input samples, and lower for highly variable parameters.
- Denoting $g_{t,i} = \nabla_{\theta_i} L(\theta)$, we get

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i},$$

where G_t is a diagonal matrix with $G_{t,ii} = G_{t-1,ii} + g_{t,i}^2$ that accumulates the total gradient value over learning history.

- So learning rate always goes down, but at different rates for different θ_i .

- One problem: G keeps increasing, and learning rate sometimes decreases too rapidly.
- *Adadelta* – same idea, but gradient history is computed with decay:

$$G_{t,ii} = \rho G_{t-1,ii} + (1 - \rho) g_{t,i}^2.$$

- The rest is the same:

$$u_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}} \mathbf{g}_{t-1}.$$

- *Adam*: also a modification of AdaGrad, uses smoothed versions of the average and mean squared gradients:

$$m_t = \beta_1 m + (1 - \beta_1) g_t,$$

$$v_t = \beta_2 m + (1 - \beta_2) g_t^2,$$

$$u_t = \frac{\eta}{\sqrt{v_t + \epsilon}} m_t.$$

- By now Adam is the method of choice.

Thank you for your attention!