{Lecture 22}

Training neural networks

Minimize $\mathcal{J}(\vec{\theta}) = -\sum_{n=1}^{N} \log p(\vec{y}_n | \vec{x}_n ; \vec{\theta})$

$\vec{\theta}$: NN weights + biases

+ regularizer [optionally]

→ no convexity guarantees ←

(1.) Vanishing and exploding gradients

Consider $\vec{g}_\ell = \frac{\partial \mathcal{J}}{\partial \vec{z}_\ell} = \frac{\partial \mathcal{J}}{\partial \vec{z}_{\ell+1}} \frac{\partial \vec{z}_{\ell+1}}{\partial \vec{z}_\ell} = J_\ell \vec{g}_{\ell+1}$

backward pass

$\vec{z}_\ell$: activations in layer ℓ

J_ℓ: Jacobian matrix

Thus $\vec{g}_\ell = J_\ell J_{\ell+1} \cdots J_{L-1} \vec{g}_L$

$\vec{g}_L$: grad at the final layer

$J_\ell \cdots J_{L-1}$: may amplify or decrease the grad depending on the eigenvalues of the Jacobian matrices:

$|\lambda_{max}^{(i)}|$ = spectral radius of J_i

→ Exploding grad ⇒ grad clipping:

$\vec{g} \to \vec{g}' = \min(1, \frac{c}{|\vec{g}|}) \vec{g}$,

c = hyperprm.

→ Vanishing grad ⇒ modify activation functions (A) or NN architecture (B)

(A) Use non-saturating activation functions such as ReLU:

$$\mathrm{ReLU}(a) = a\,\mathbb{I}(a>0)$$

However, it's possible to have "the dead ReLU problem":

some components of $\vec{a}_\ell = W\vec{z}_{\ell-1}$ may become negative $\Rightarrow$ the corresp. components of $\vec{z}_\ell = \sigma(\vec{a}_\ell)$ will be set to 0, and the units will stay off.

This problem can be solved by using the leaky ReLU:

neg. inputs ↓ pos. inputs ↓

$$\mathrm{LReLU}(a;\alpha) = \max(\alpha a, a),$$

$0<\alpha<1$ is a hyperprm that can be learned

Another choice is ELU:

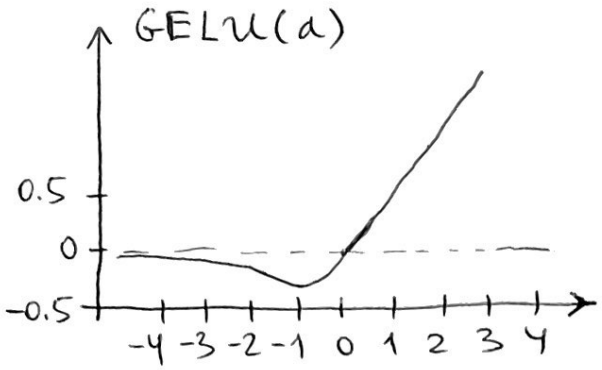
$$\mathrm{ELU}(a;\alpha) = \begin{cases} \alpha(e^a - 1) & a \le 0 \\ a & a > 0 \end{cases}$$

↖ smooth f'n

Finally, GELU (gaussian error linear unit) is popular:

$$\mathrm{GELU}(a) = a\,\Phi(a), \text{ where}$$

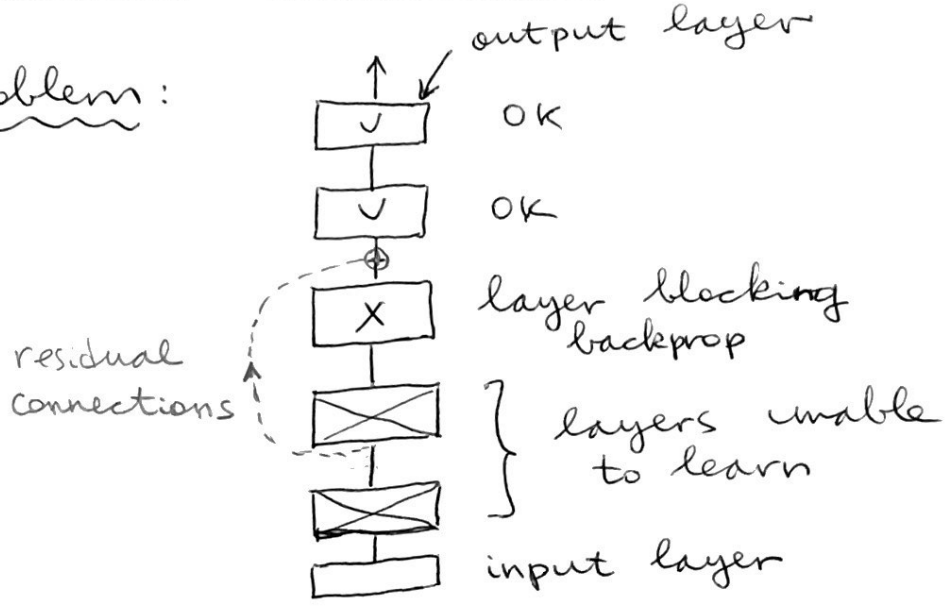
$$\Phi(a) = \Pr(\mathcal{N}(0,1) \le a) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{a}{\sqrt{2}}\right)\right)$$



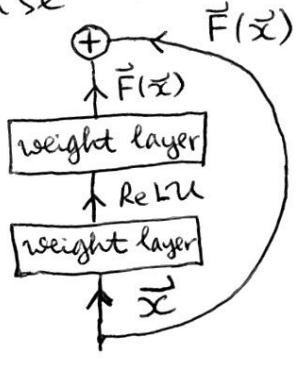
non-monotonic

(B) Residual connections (ResNet)

Problem:



Idea: use residual blocks



later layers directly coupled to earlier layers through these "skip connections"

(2.) Parameter initialization

Consider $a_i = \sum_{j=1}^{n_{in}} w_{ij} x_j$, where

n_{in} = # of incoming connections.

If $w_{ij} \sim \mathcal{N}(0, \sigma^2)$ and

$\begin{cases} E[x_j] = 0, \\ Var[x_j] = \gamma^2, \end{cases}$ we obtain:

$$E[a_i] = \sum_{j=1}^{n_{in}} E[w_{ij}] E[x_j] = 0,$$

↑ indep. assumption

$$Var[a_i] = E[a_i^2] - \underbrace{(E[a_i])^2}_{=0} =$$

$$= E\left[\sum_{j,k=1}^{n_{in}} w_{ij} w_{ik} x_j x_k\right] = \sum_{j=1}^{n_{in}} E[w_{ij}^2 x_j^2] \quad (=)$$

↑ $j \neq k$ terms do not contribute

$$(=) \sum_j E[w_{ij}^2] E[x_j^2] = n_{in} \sigma^2 \gamma^2.$$

(indep.)

Need to set $n_{in} \sigma^2 = 1$, otherwise

$Var[a_i] \nearrow$ as $n_{in} \nearrow$.

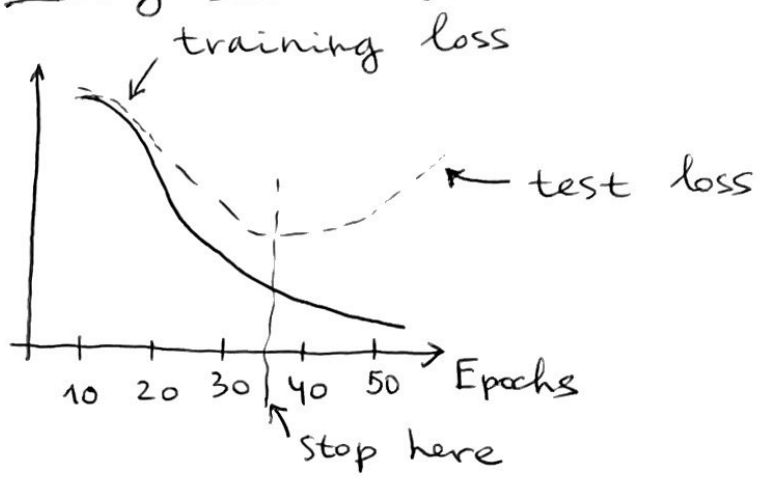
Likewise, in the backwards pass the Var of the gradients may blow up unless $n_{out} \sigma^2 = 1$, where n_{out} = # of outgoing connections.

One can set $\sigma^2 = \frac{2}{n_{in} + n_{out}}$

Xavier initialization

(3.) Regularization

(1.) Early stopping.



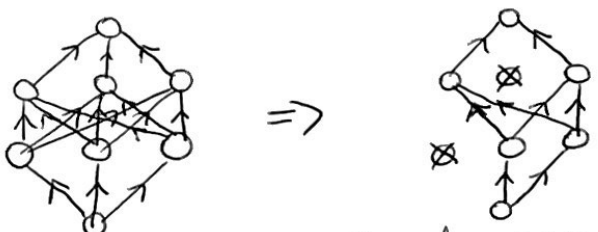
(2.) Weight decay.

add gaussian priors for weights and biases: $\begin{cases} \mathcal{N}(\vec{w} \mid \vec{0}, \alpha^2 \mathbb{I}) \\ \mathcal{N}(\vec{b} \mid \vec{0}, \beta^2 \mathbb{I}) \end{cases}$

L_2 regularization; encourages smaller prm values

(3.) Dropout

Turn off neurons with prob. P



"each neuron learns to perform by itself, even if other neurons are missing"

Using dropout can be viewed as estimating the noisy versions of the weights:

$\theta_{ij} = w_{ij}\,\epsilon$, where

(θ_{ij} → estimated during training)

$\epsilon = \text{Ber}(1-p)$

$\begin{cases} i = \text{unit } i, \text{ layer } l-1 \\ j = \text{all units in layer } l \text{ connected to node } i \end{cases}$

$$E[\theta_{ij}] = w_{ij}(1-p) + 0 \times p .$$

At testing, remove the dropout and rescale the weights:

$$\underline{w_{ij} = \frac{\theta_{ij}}{1-p}} .$$

Finally, one can use dropout to create an ensemble of networks ⇒ [Monte Carlo dropout]

$$p(\vec{y}|\vec{x},\mathcal{D}) = \frac{1}{S}\sum_{k=1}^{S} p(\vec{y}|\vec{x}; N_k)$$

(S ↑ # dropout networks; N_k ↑ labels network k topology & weights)