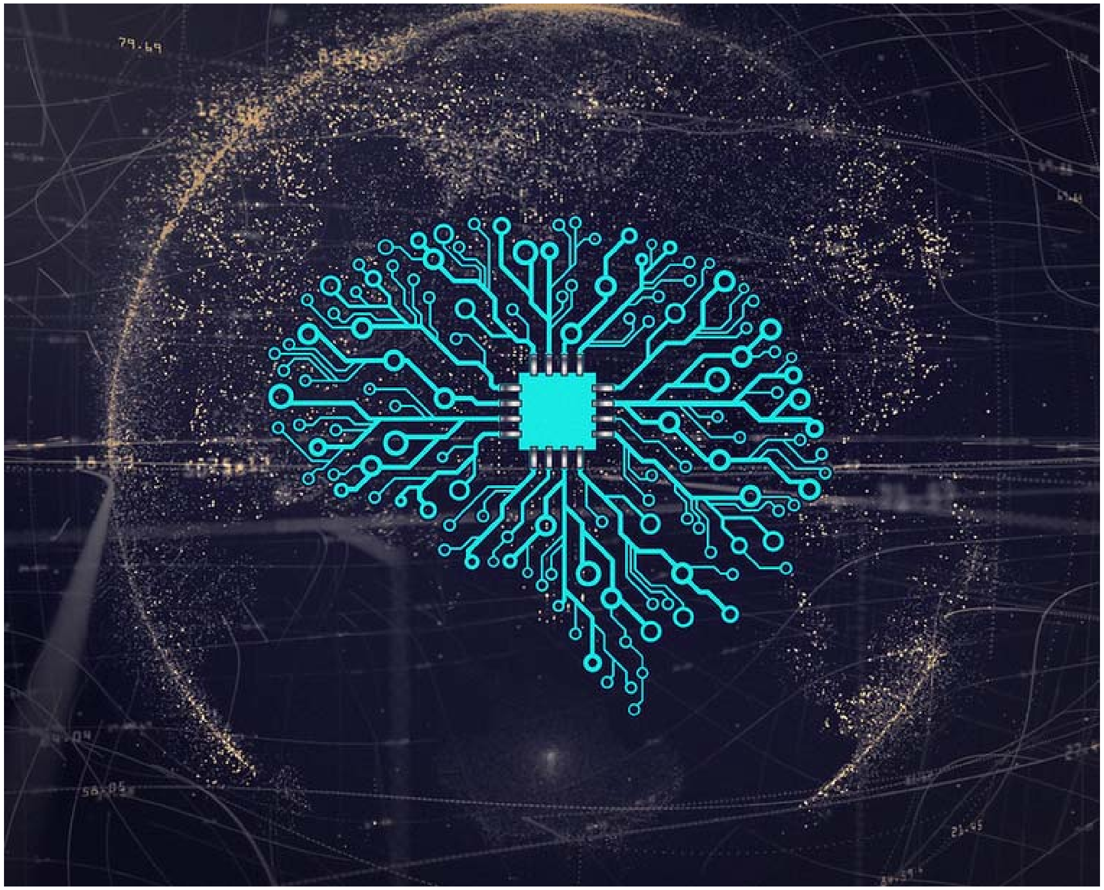
Neural Networks Tricks and Tips

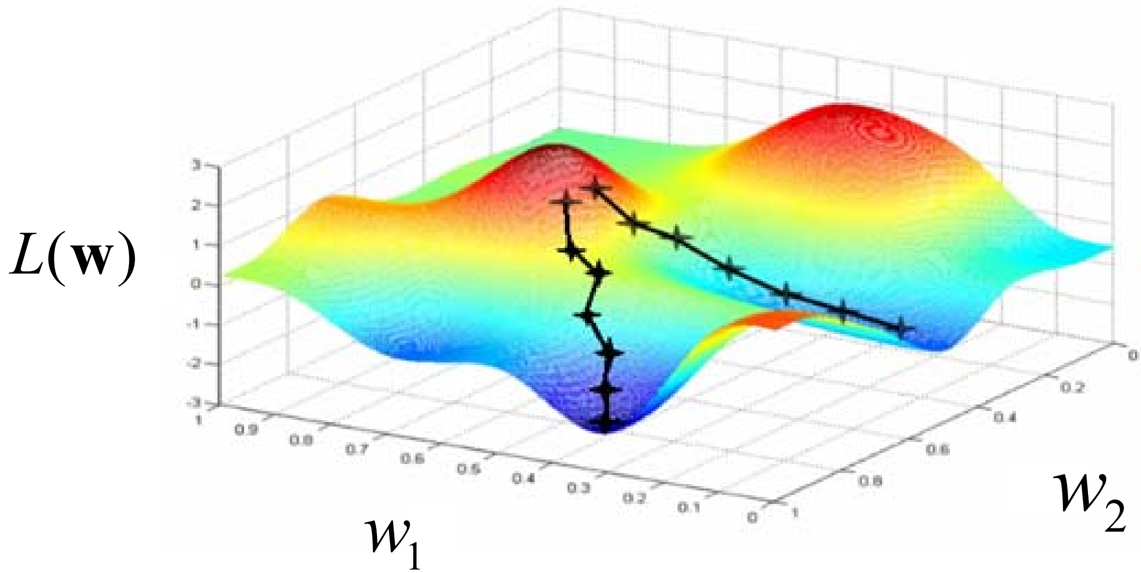


Intelligent Visual Computing
Evangelos Kalogerakis

Gradient Descent

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \eta \frac{\sum_{i \in R} \nabla_{\mathbf{w}} L_i(\mathbf{w})}{|R|}$$

R is a random minibatch of training examples,
L(**w**) is the loss wrt NN parameters**, η** is the learning rate



Lots and lots of local minima in neural networks!

Yet, this does not work so easily…

Yet, this does not work so easily...

- Optimization becomes difficult with **many layers**.

- Hard to diagnose and **debug malfunctions**.

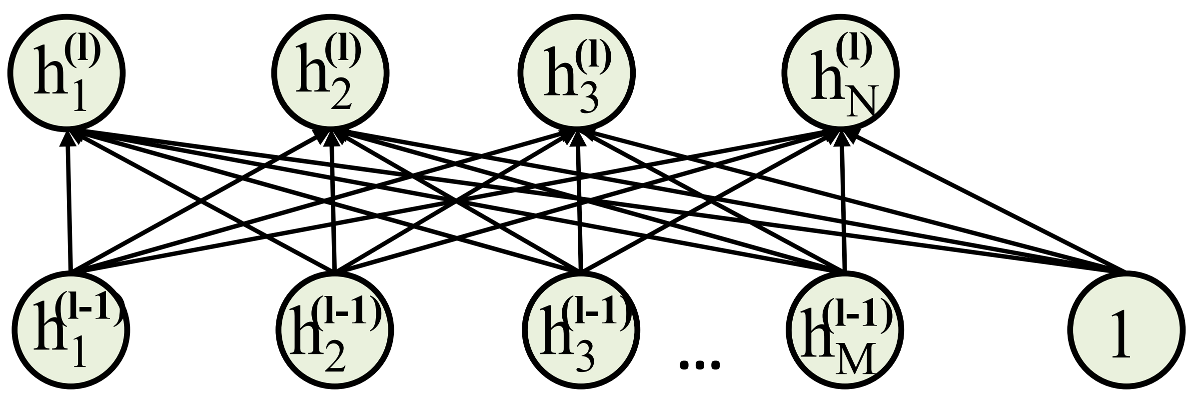
- **Many things turn out to matter**:
  - Initialization of parameters
  - Optimization procedure and hyper-parameters (step size)
  - Network structure

Initialization

Initialize filters/weights to small values. How "small" should these be?

Assume one linear layer with output: $h_n^{(l)} = \mathbf{w}_n \bullet \mathbf{h}^{(\mathbf{l-1})}$

$$Var[h_n^{(l)}] = M \cdot Var[w_{n,m}^{(l)}]Var[h_m^{(l-1)}]$$



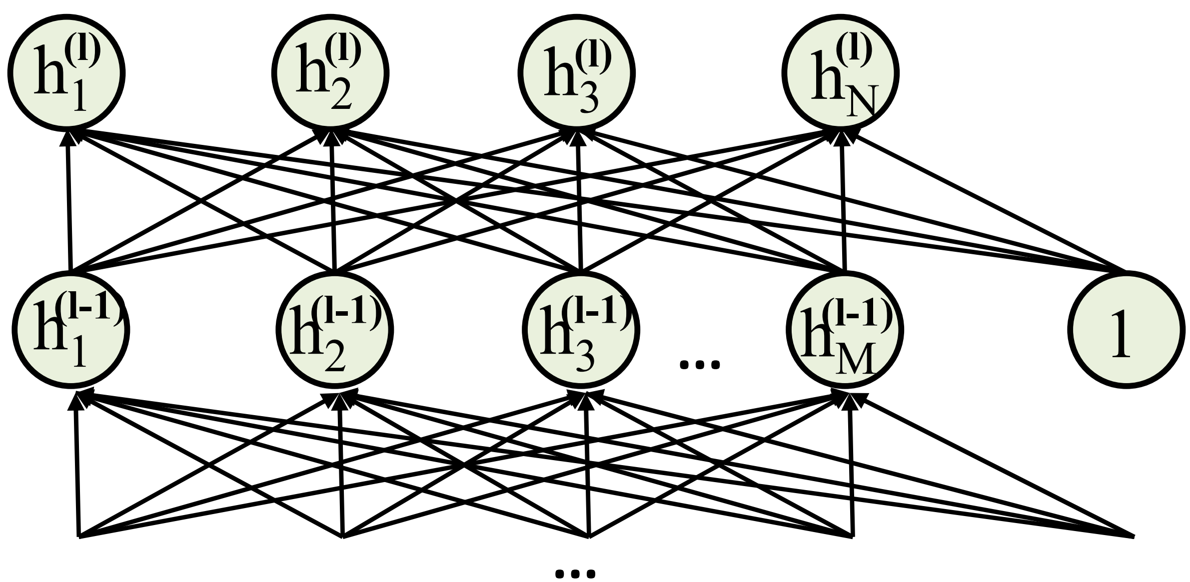
Sketch of a proof:
http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization

Initialization

OK, but how "small" these random values should be?

The input to this layer depends on many previous layers...

$$Var[h_n^{(l)}] = \Big(\prod_{\substack{layer \\ b=1\ldots l-1}} M^{(b)} \cdot Var[w_{n,m}^{(b)}]\Big) Var[input]$$



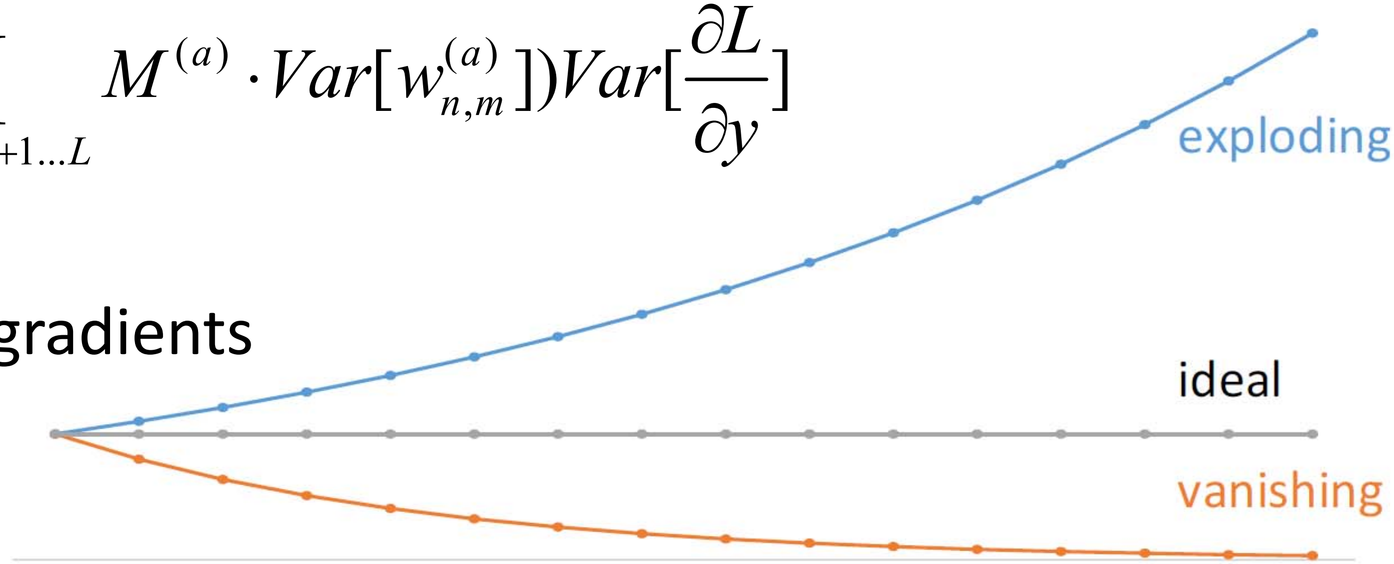
Initialization

Similar behavior for gradients!

$$Var[h_n^{(l)}] = \Big(\prod_{layer\ b=1\ldots l-1} M^{(b)} \cdot Var[w_{n,m}^{(b)}]\Big) Var[input]$$

$$Var\Big[\frac{\partial L}{h_n^{(l)}}\Big] = \Big(\prod_{layer\ a=l+1\ldots L} M^{(a)} \cdot Var[w_{n,m}^{(a)}]\Big) Var\Big[\frac{\partial L}{\partial y}\Big]$$

Both outputs & gradients can easily **explode** or **vanish!!!**



Proof sketch: http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization

Initialization

Want: $$M \cdot Var[w_{n,m}] = 1$$ $$N \cdot Var[w_{n,m}] = 1$$ (M input nodes, N output nodes)

A trade-off: $$Var[w_{n,m}] = \frac{2}{N+M}$$

Gaussian dist. Initialization: $N(0,r^2)$ $$r = \sqrt{\frac{2}{N+M}}$$

Uniform dist. Initialization: [-r, r] $$r = \sqrt{\frac{6}{N+M}}$$

[Understanding the difficulty of training deep feedforward neural networks, Glorot & Bengio 2010]

Initialization (for ReLUs)

Biases are often set to 0 or small positive numbers e.g., 0.01 (to prevent ReLUs to get stuck at their negative part).

For the rest of the weights:

Gaussian dist. Initialization: N(0,r^2)

$$Var[w_{n,m}] = \frac{2}{N} \;\; or \;\; \frac{2}{M} \;\; or \;\; \frac{4}{N+M}$$

Uniform dist. initialization [-r, r]

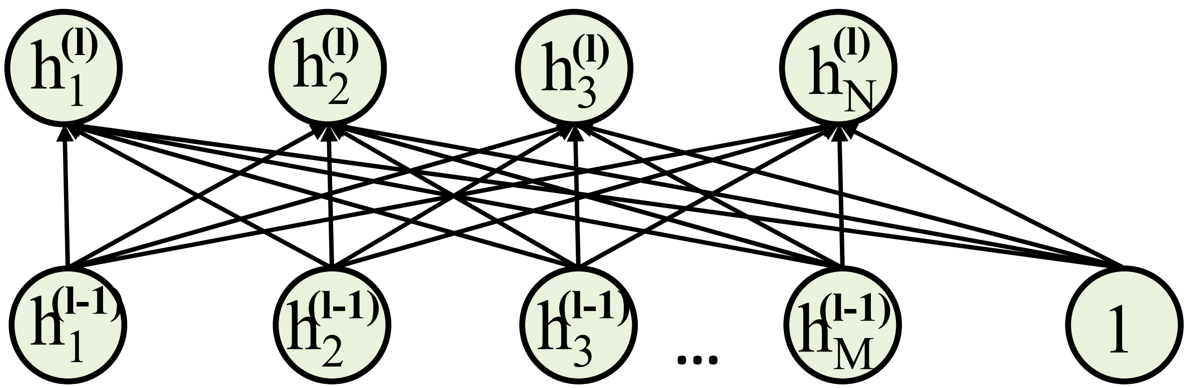
$$r = \sqrt{\frac{6}{N}} \;\; or \;\; \sqrt{\frac{6}{M}} \;\; or \;\; \sqrt{\frac{12}{N+M}}$$

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,He et al.

Batch Normalization

During training, weights will change, variances will change, distributions of layer outputs can vary wildly!

$$Var[h_n^{(l)}] = M \cdot Var[w_{n,m}^{(l)}]Var[h_m^{(l-1)}]$$

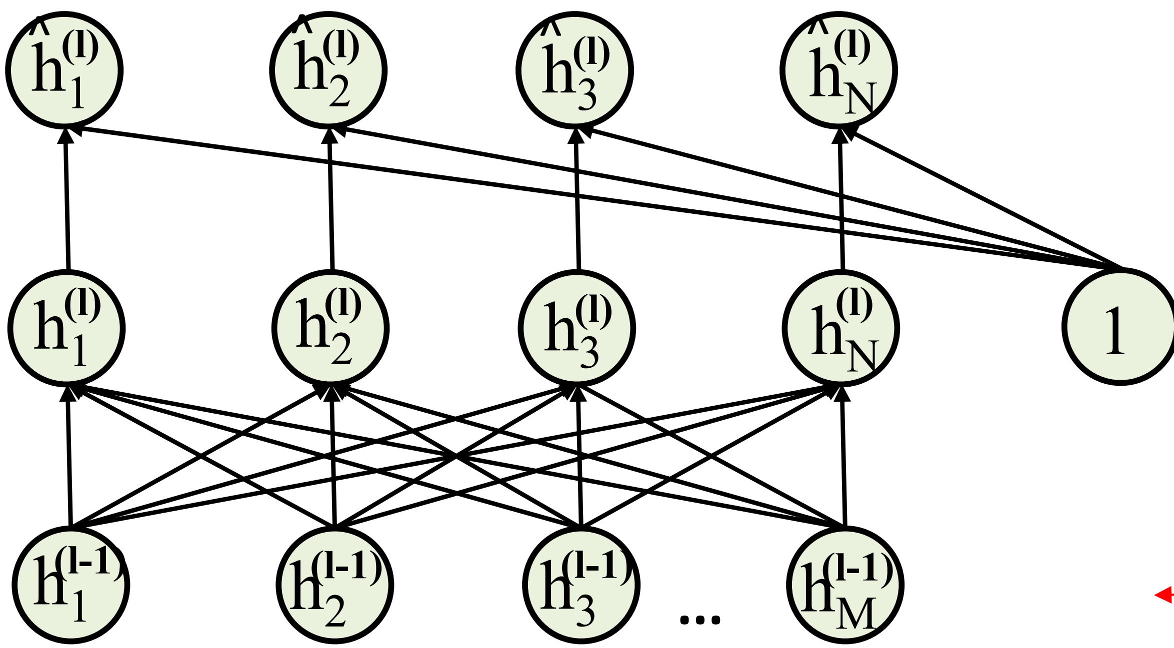


Let’s explicitly fix the distributions of our nodes => Batch normalization

Batch Normalization

Standardize outputs within each batch, then learn to scale & shift them:

$$\hat{h}_n^{(l)} = \frac{h_n^{(l)} - E_{batch}[h_n^{(l)}]}{\sqrt{Var_{batch}[h_n^{(l)}] + \varepsilon}} \qquad \hat{h}_n^{(l)} = \gamma_n^{(l)} \hat{h}_n^{(l)} + \beta_n^{(l)}$$



Practically implemented as another layer!

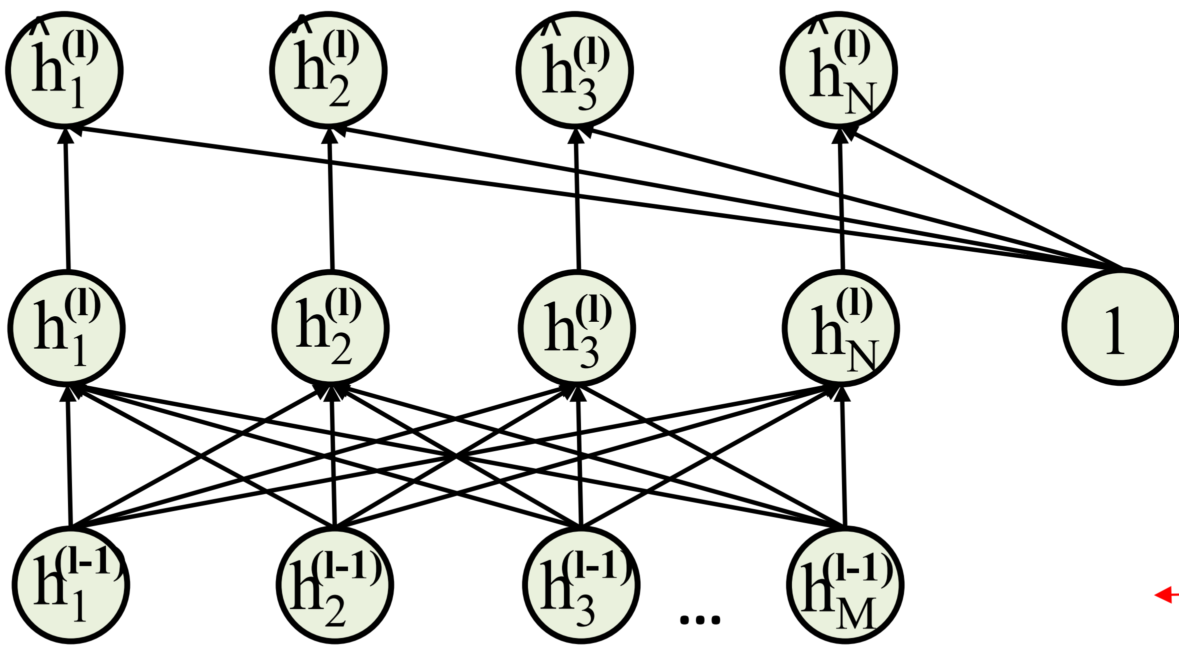
Parameters γ, β per node and learned

Bias = β
(no need to use a bias here)

Batch Normalization

Standardize outputs within each **batch**, then learn to scale & shift them:

$$\hat{h}_n^{(l)} = \frac{h_n^{(l)} - E_{batch}[h_n^{(l)}]}{\sqrt{Var_{batch}[h_n^{(l)}] + \varepsilon}} \qquad \hat{h}_n^{(l)} = \gamma_n^{(l)} \hat{h}_n^{(l)} + \beta_n^{(l)}$$



Practically implemented as another layer!

Parameters γ, β per node and learned

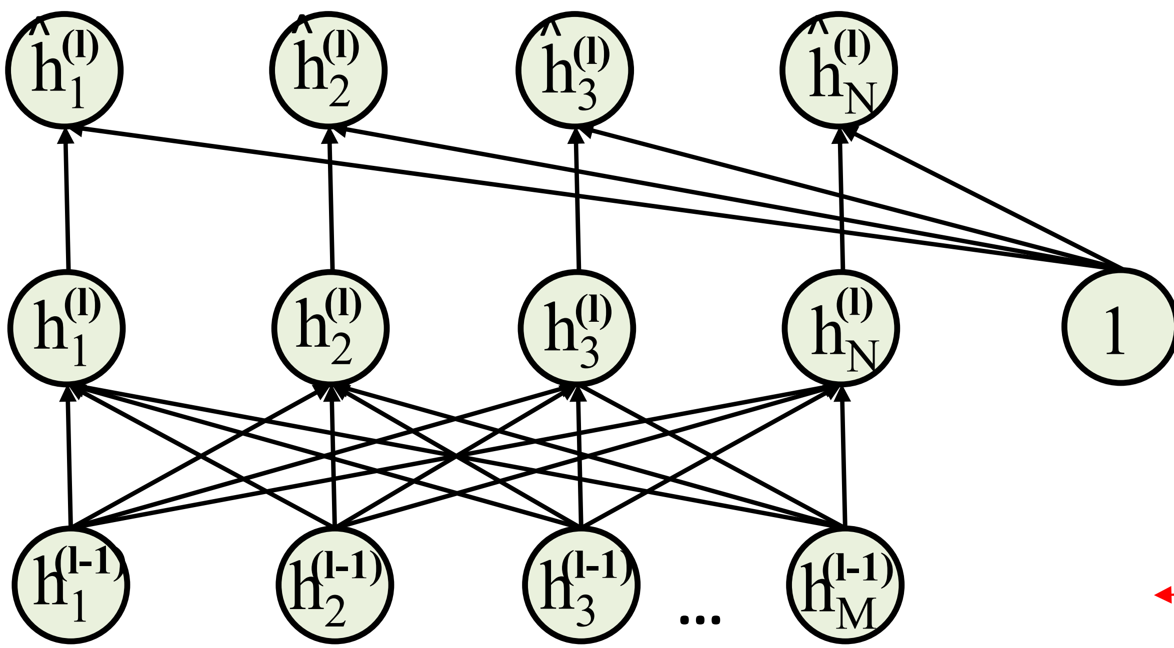
Bias = β
(no need to use a bias here)

... Yet, when batch size is 1, we can't compute the above statistics (or are unreliable for small batches!)

Layer Normalization

Standardize outputs within each **layer**, then learn to scale & shift them:

$$\hat{h}_n^{(l)} = \frac{h_n^{(l)} - E_{layer}[h^{(l)}]}{\sqrt{Var_{layer}[h^{(l)}] + \varepsilon}} \qquad \hat{h}_n^{(l)} = \gamma_n^{(l)} \hat{h}_n^{(l)} + \beta_n^{(l)}$$



Practically implemented as another layer!

Parameters γ, β per node and learned

Bias = β
(no need to use a bias here)

See also Group Normalization: https://arxiv.org/pdf/1803.08494.pdf

Momentum + regularization

Modify stochastic/batch gradient descent:

$$\text{Before}: \quad \Delta\mathbf{w} = \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad \mathbf{w} = \mathbf{w} - \Delta\mathbf{w}$$

$$\text{With momentum}: \Delta\mathbf{w} = \mu \Delta\mathbf{w}_{previous} + \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad \mathbf{w} = \mathbf{w} - \Delta\mathbf{w}$$

"Smooth" estimate of gradient from iterations:

- High-curvature directions cancel out, low-curvature directions "add up" & accelerate. Often set to with μ=0.9

Momentum + regularization

Modify stochastic/batch gradient descent:

$$\text{Before}: \quad \Delta\mathbf{w} = \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad \mathbf{w} = \mathbf{w} - \Delta\mathbf{w}$$

$$\text{With momentum}: \Delta\mathbf{w} = \mu \Delta\mathbf{w}_{previous} + \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad \mathbf{w} = \mathbf{w} - \Delta\mathbf{w}$$

"Smooth" estimate of gradient from iterations:

- High-curvature directions cancel out, low-curvature directions "add up" & accelerate. Often set to with μ=0.9

Use **weight decay** to discourage large weights:

$$\mathbf{w} = \mathbf{w} - \Delta\mathbf{w} - \eta\lambda\mathbf{w}$$

Related to adding a penalty to the loss: $L(\mathbf{w}) + \frac{1}{2}\lambda \|\mathbf{w}\|^2$

Momentum + regularization

Modify stochastic/batch gradient descent:

$$\text{Before:} \quad \Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad w = w - \Delta \mathbf{w}$$

$$\text{With momentum:} \ \Delta \mathbf{w} = \mu \Delta \mathbf{w}_{previous} + \eta \nabla_{\mathbf{w}} L(\mathbf{w}), \quad w = w - \Delta \mathbf{w}$$

"Smooth" estimate of gradient from iterations:

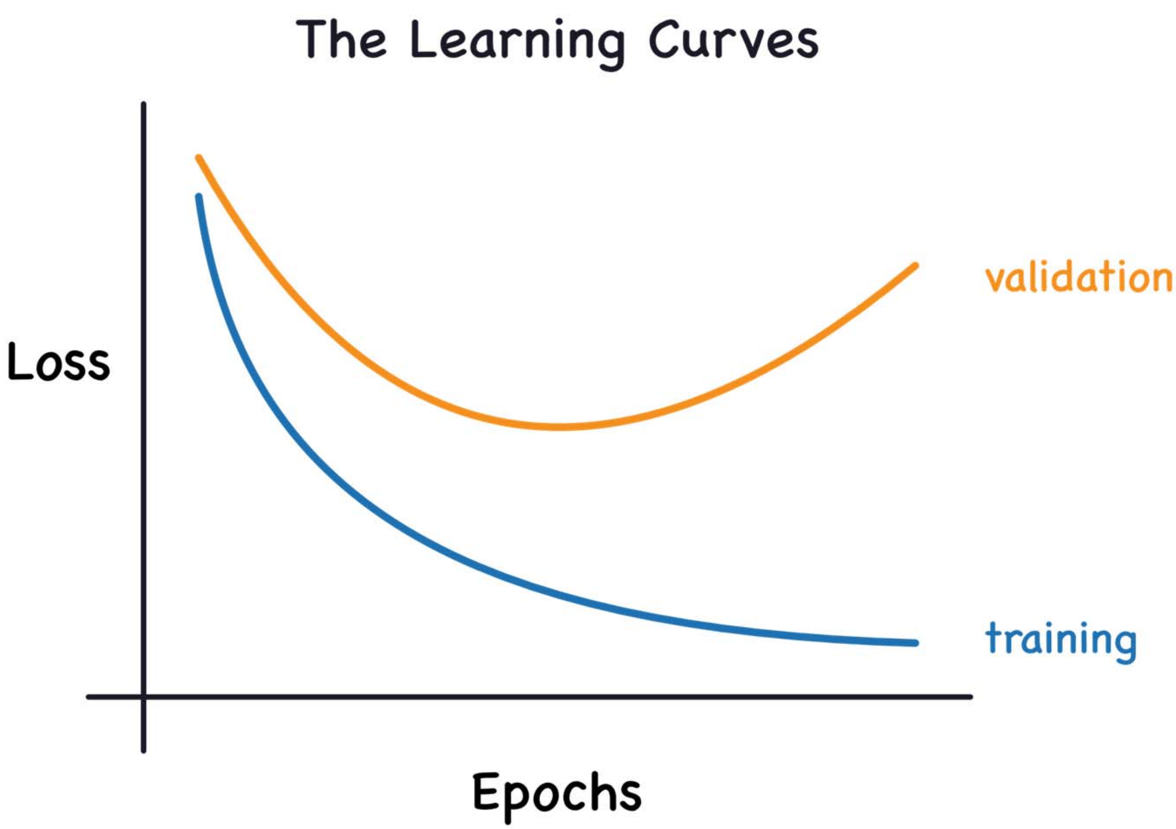
- High-curvature directions cancel out, low-curvature directions "add up" & accelerate. Often set to with μ=0.9

- **Other SGD variants:** RMSprop, Adam, AdamW

See also:
https://en.wikipedia.org/wiki/Stochastic_gradient_descent
https://ruder.io/optimizing-gradient-descent/
https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

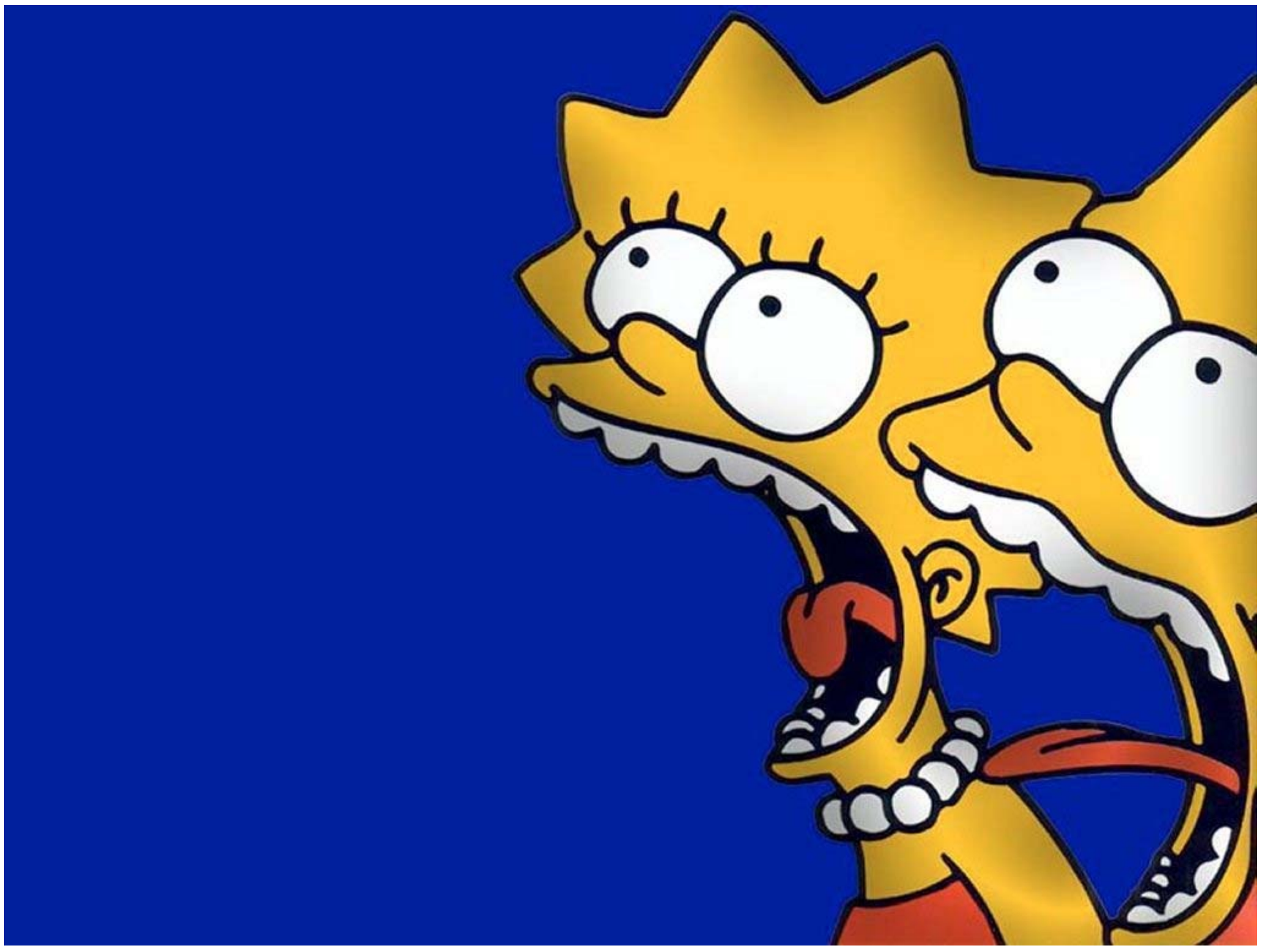
Baby-sitting

Some **baby-sitting** is necessary! Track loss function during training and also check loss / accuracy in the validation set at the same time!



https://www.kaggle.com/code/ryanholbrook/overfitting-and-underfitting/tutorial

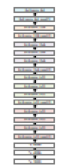
Yet, things will not still work well!



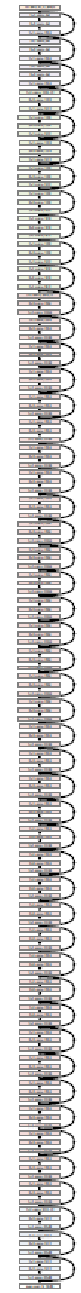
Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)

VGG, 19 layers
(ILSVRC 2014)

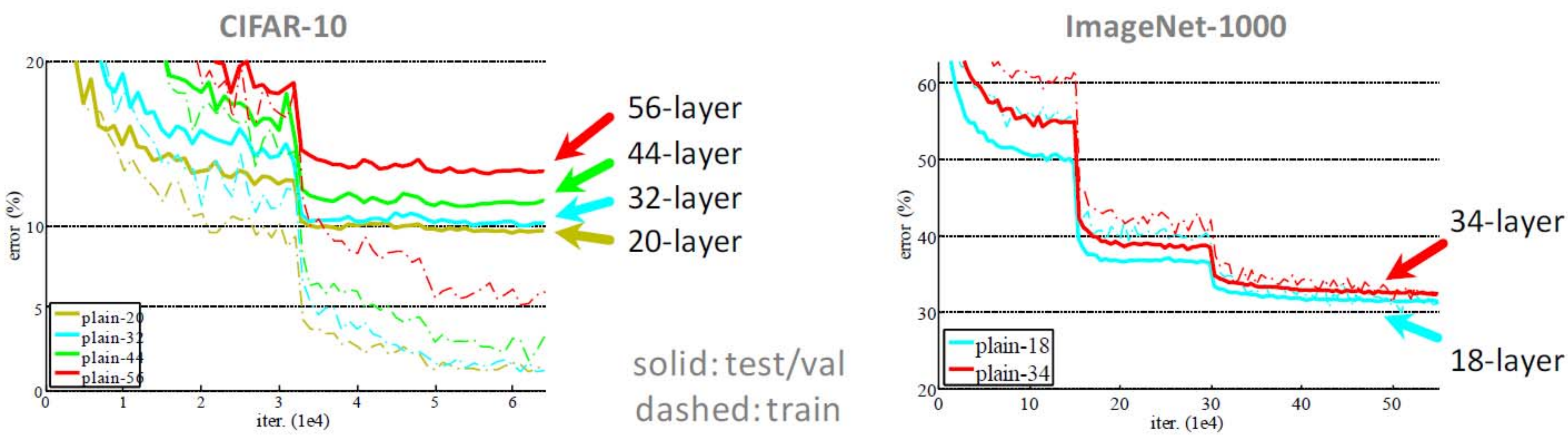


ResNet, 152 layers
(ILSVRC 2015)



Is learning better networks as simple as stacking more layers?

The deeper, the better?

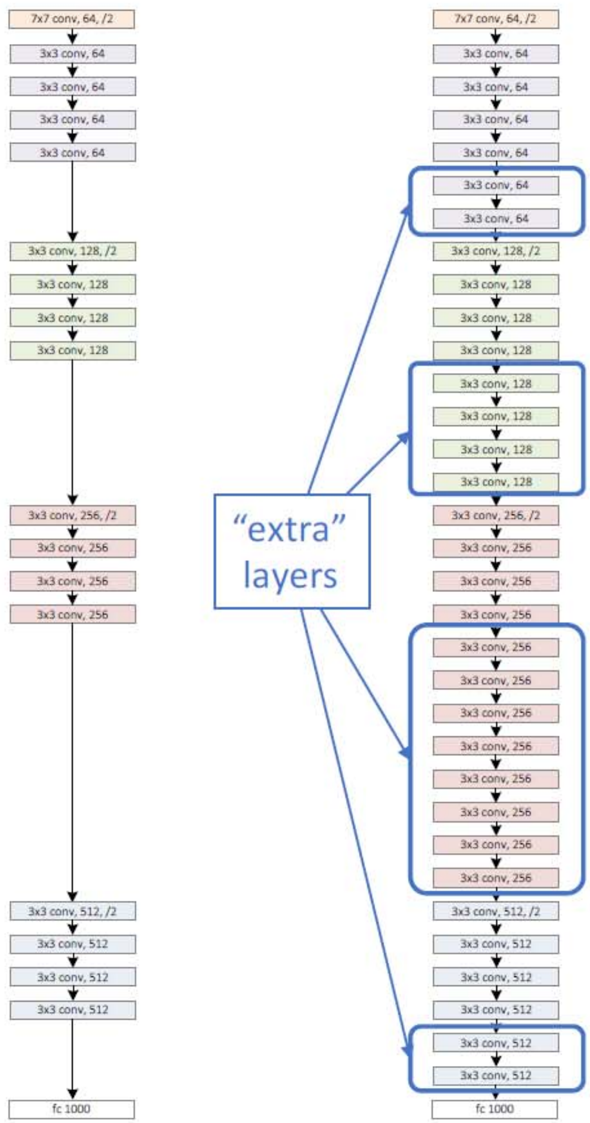


Stacking more layers in "plain" nets results in **higher training error (and test error)**

A general phenomenon, observed in many datasets

ResNets basic idea

a shallower model (18 layers)



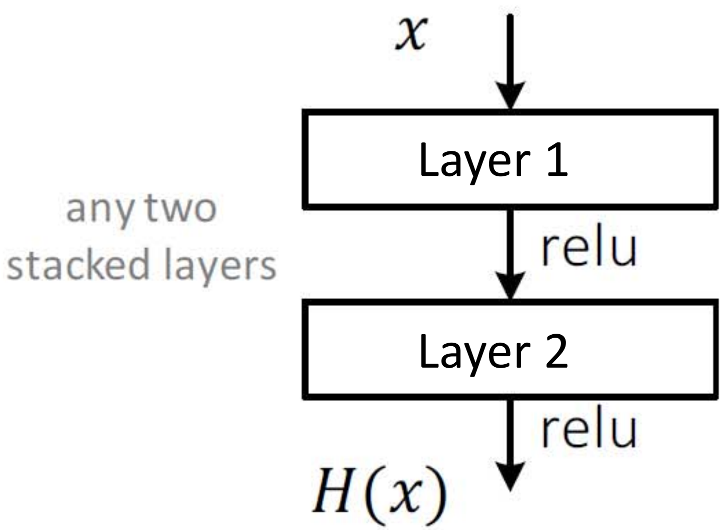
a deeper counterpart (34 layers)

- Richer solution space
- A deeper model should not have **higher training error**
- A solution *by construction*:
  - original layers: copied from a learned shallower model
  - extra layers: set as identity
  - at least the same training error

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

ResNets basic idea

- Plain net



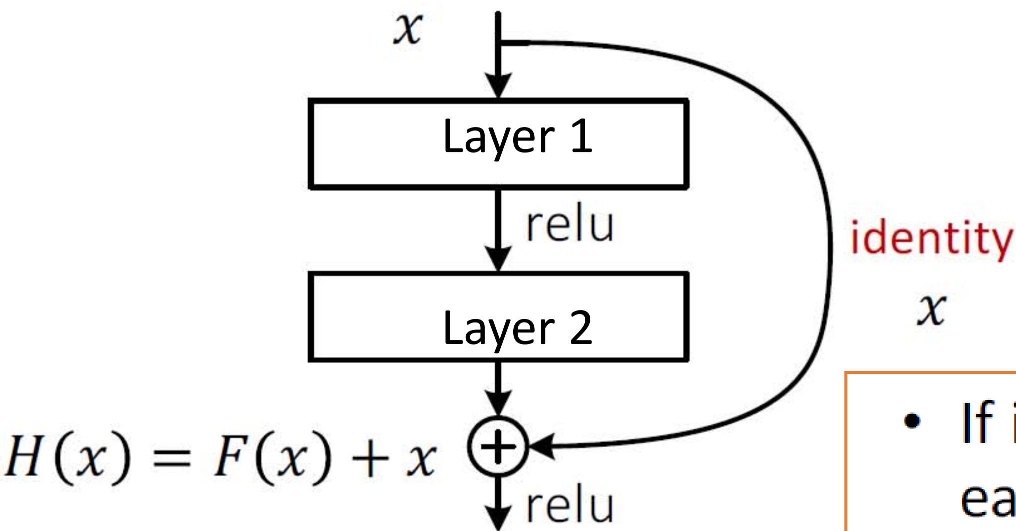
$H(x)$ is any desired mapping,

hope the 2 weight layers fit $H(x)$

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

ResNets basic idea

- Residual net



$H(x)$ is any desired mapping,

~~hope the 2 weight layers fit~~ $\cancel{H(x)}$

hope the 2 weight layers fit $F(x)$

let $H(x) = F(x) + x$

- If identity were optimal,
  easy to set weights as 0

- If optimal mapping is closer to identity,
  easier to find small fluctuations

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.