

Deep Learning (BEV033DLE)

Lecture 6 Weight initialisation, batch normalisation, Resnets

Czech Technical University in Prague

- ◆ Weight initialisation
- ◆ Batch normalisation
- ◆ Residual neural networks
- ◆ Transfer learning

Weight initialisation

(1) Initialising all weights and biases with zero is not a good idea. Why?

Side step: symmetries and gradients:

Consider a scalar function $f(w)$ that is invariant to the linear mapping $B: \mathbb{R}^n \rightarrow \mathbb{R}^n$, i.e. $f(Bw) = f(w)$. Its gradient ∇f has the property

$$\nabla f(Bw) = B^{-T} \nabla f(w),$$

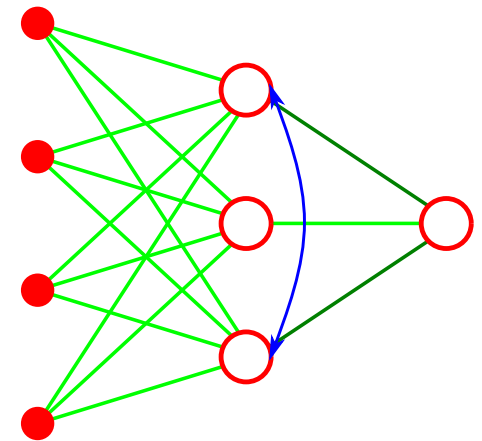
which follows from

$$\langle \nabla f(Bw), u \rangle := \lim_{t \downarrow 0} \frac{f(Bw + tu) - f(Bw)}{t} \stackrel{!}{=} \langle \nabla f(w), B^{-1}u \rangle$$

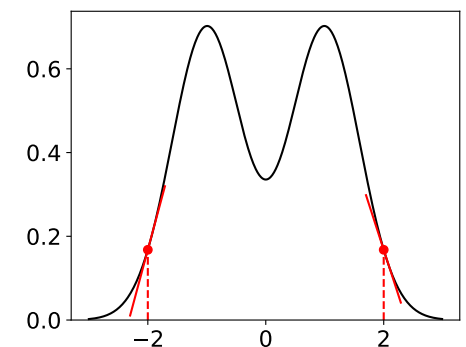
What happens if SGD is started from an invariant point $w_0 = Bw_0$ and $B^{-T} = B$ holds?

$$B[w_0 - \alpha \nabla f(w_0)] = w_0 - \alpha \nabla f(Bw_0) = w_0 - \alpha \nabla f(w_0)$$

The new point w_1 will be again invariant, i.e. $Bw_1 = w_1$.
 We need to break the symmetry!



Invariance
w.r.t. permutations

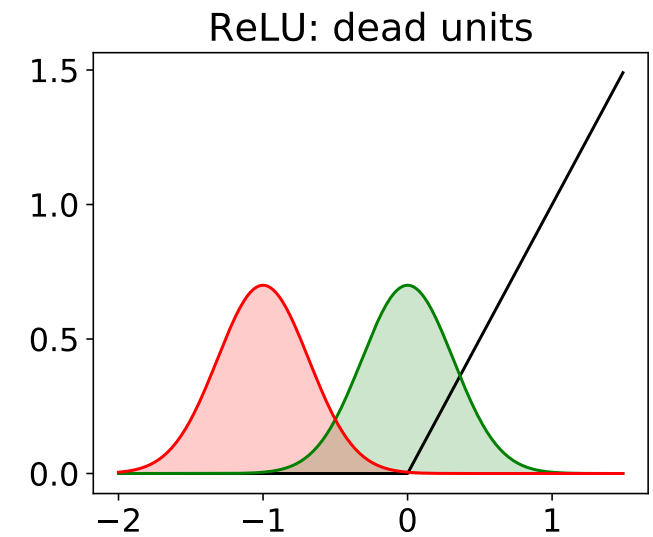
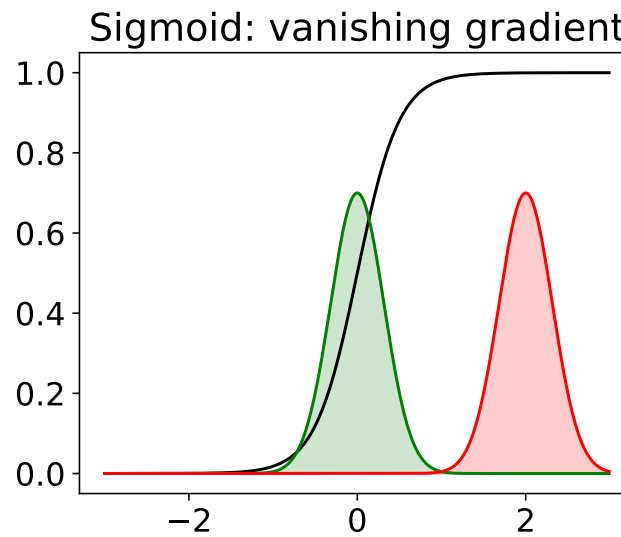
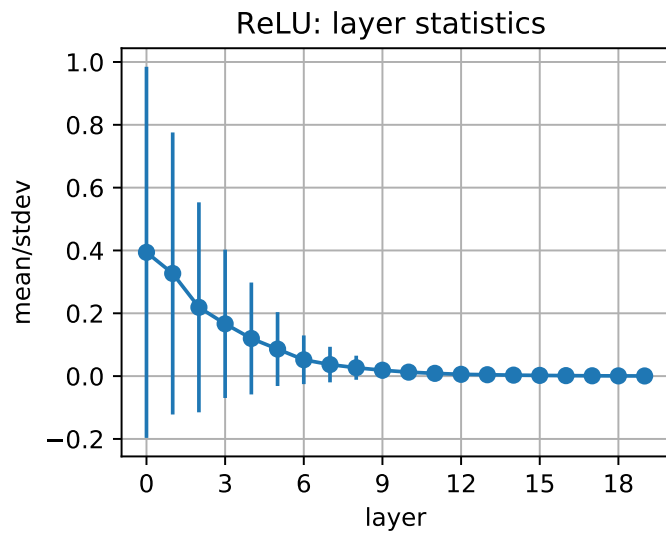


Gradient of a mirror
symmetric function

Weight initialisation

(2) Initialise all weights and biases randomly from a uniform (or normal) distribution.

- ◆ o.k. for shallow networks,
- ◆ not o.k. for deep networks!



Left: node statistics for the layers of a deep FFN with ReLU units, all weights initialised from a normal distribution.

Middle and right: this can lead to vanishing/exploding gradients and “dead units” during learning

Weight initialisation

(3) **Proper initialisation:** Initialise weights/biases so that each neuron has activation statistic (over the dataset) with certain mean and variance.

Example 1 (Glorot & Bengio, 2010). Analyse variance of neuron outputs and backprop gradients under the following simplifying assumptions

- ◆ Tanh activation function $f(x)$ in linear regime, i.e, $f(x) \approx x$
- ◆ Neuron outputs as well as gradient components are i.i.d.

Start from a single neuron $y = w^T x$, $x \in \mathbb{R}^n$. Assume

- ◆ x_i are i.i.d. with $\mathbb{E}[x_i] = 0$ and $\mathbb{V}[x_i] = \chi$
- ◆ w_i are i.i.d. with $\mathbb{E}[w_i] = 0$ and $\mathbb{V}[w_i] = \omega$

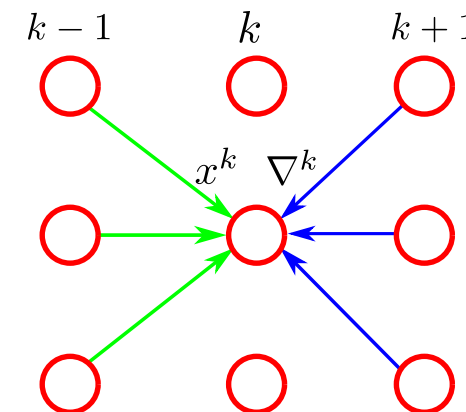
It follows that $\mathbb{E}[y] = 0$ and $\mathbb{V}[y] = n\omega\chi$.

Consider now a feedforward network with Tanh activation and assumptions as above. For layer k with n_k nodes, denote neuron outputs by x^k and gradients by ∇^k . Denote the variance of weights in layer k by ω_k .

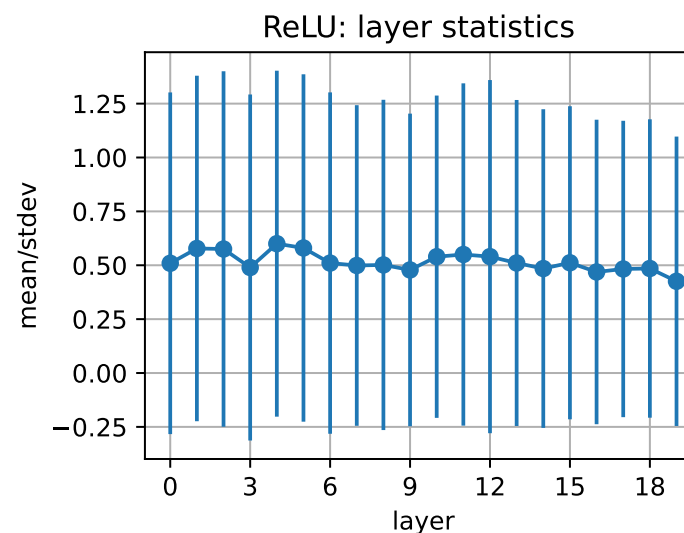
Weight initialisation

Example 1 (cont.).

- ◆ forward: $\mathbb{V}[x_i^k] = n_{k-1}\omega_k \mathbb{V}[x_j^{k-1}]$
 We want $\mathbb{V}[x_i^k] \approx \mathbb{V}[x_j^{k-1}]$, i.e. $n_{k-1}\omega_k = 1$.
- ◆ backward: $\mathbb{V}[\nabla_i^k] = n_{k+1}\omega_{k+1} \mathbb{V}[\nabla_j^{k+1}]$
 We want $\mathbb{V}[\nabla_i^k] \approx \mathbb{V}[\nabla_j^{k+1}]$, i.e. $n_k\omega_k = 1$
- ◆ Compromise: Set $\omega_k = \frac{2}{n_{k-1} + n_k}$. Assuming that the inputs x^0 have zero mean and unit variance, initialise the weights randomly by $w_{ij}^k \sim \mathcal{N}(0, \omega_k)$.



Similar considerations for ReLU activation lead to a different scheme (He et al., 2015). Figure: Node statistics for the layers of the same deep FFN with ReLU units as in Slide 3. But now with a proper weight initialisation.



Batch normalisation

(Joffe & Szegedy, 2015) Motivation:

- ◆ Keep control over neuron activation statistics during training
- ◆ Alleviate the need of specialised initialisation variants
- ◆ Regularise learning & pre-condition gradients

Batch normalisation: Denote by $\mathcal{B} \subset \mathcal{T}^m$ a mini-batch of training examples and by a_i the activation of a network unit $a_i = \sum_j w_{ij} x_j$. Re-parametrise it (stochastically) by using its statistic over mini-batches

$$\mu_{\mathcal{B}} = \mathbb{E}_{\mathcal{B}}[a_i] \quad \sigma_{\mathcal{B}}^2 = \mathbb{V}_{\mathcal{B}}[a_i]$$

$$\hat{a}_i = \frac{a_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}$$

$$a_i \leftarrow \gamma \hat{a}_i + \beta \equiv BN_{\gamma, \beta}(a_i)$$

- ◆ γ_i, β_i are learnable parameters
- ◆ $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ have to be differentiated w.r.t. network parameters
- ◆ exponentially weighted averages of $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ are kept during training and used for inference.

Batch normalisation

Technical implementation of batch normalisation in PyTorch: A layer `BatchNorm1d` that

- ◆ takes a tensor x with dimension `[batchsize, channels]` on input and returns a tensor y with same dimension on output,
- ◆ has learnable parameters γ and β for each channel (init: $\gamma = 1, \beta = 0$)
- ◆ keeps running averages of the batch statistic $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ for each channel,
- ◆ depending on its state (`train, eval`) uses either the batch statistics or the saved running averages to compute its outputs.

For convolutional networks: use the layer `BatchNorm2d`, which computes statistics over batchsize and spatial dimensions.

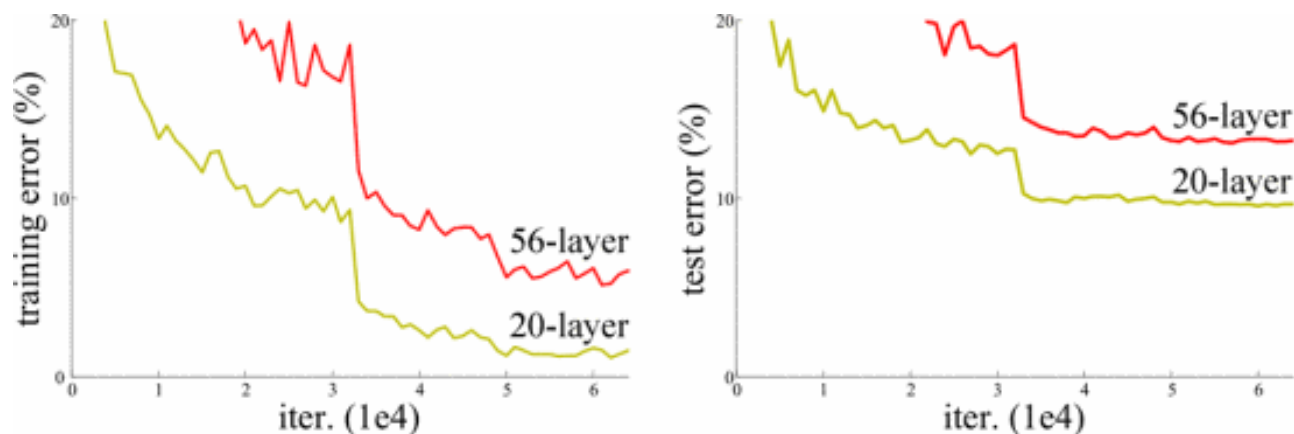
Batch normalisation:

- ◆ alleviates the need of special weight initialisation since it implements the scheme (3) discussed above for the first mini batch,
- ◆ the neuron outputs for a particular training example depend on the outputs of the other examples in the mini-batch, which in turn is stochastic.
- ◆ can be seen as stochastic re-parametrisation of weights and gradient preconditioning

$$w \rightarrow \gamma \frac{w}{\sigma_{\mathcal{B}}} \quad b \rightarrow \gamma \frac{(b - \mu_{\mathcal{B}})}{\sigma_{\mathcal{B}}} + \beta$$

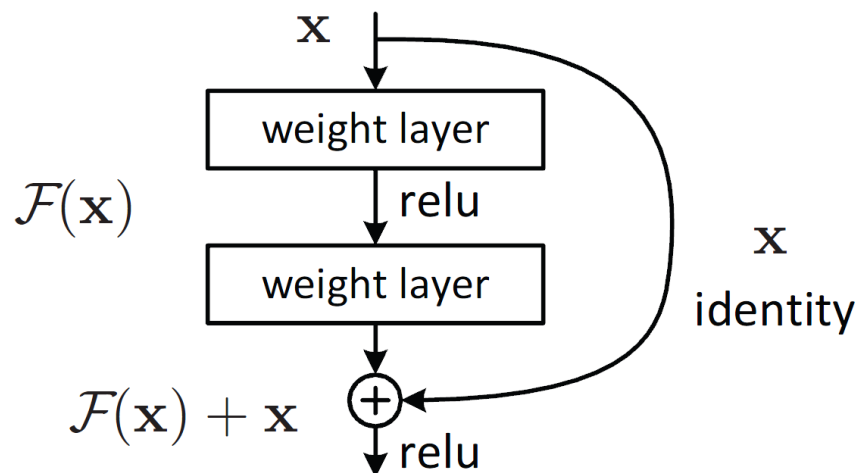
Residual Networks

By using proper weight initialisation/batch normalisation we can learn deep networks with up to 20-30 layers. Can we go for even deeper networks?

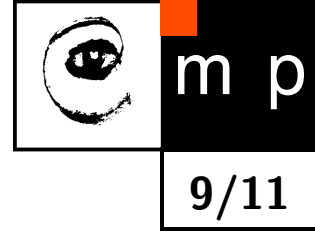


Training error and test error on CIFAR-10 with 20-layer and 56-layer “plain” networks.

He et al., CVPR 2016: Yes, by using the architecture of *residual networks*. They introduce *skip connections*.



Residual Networks



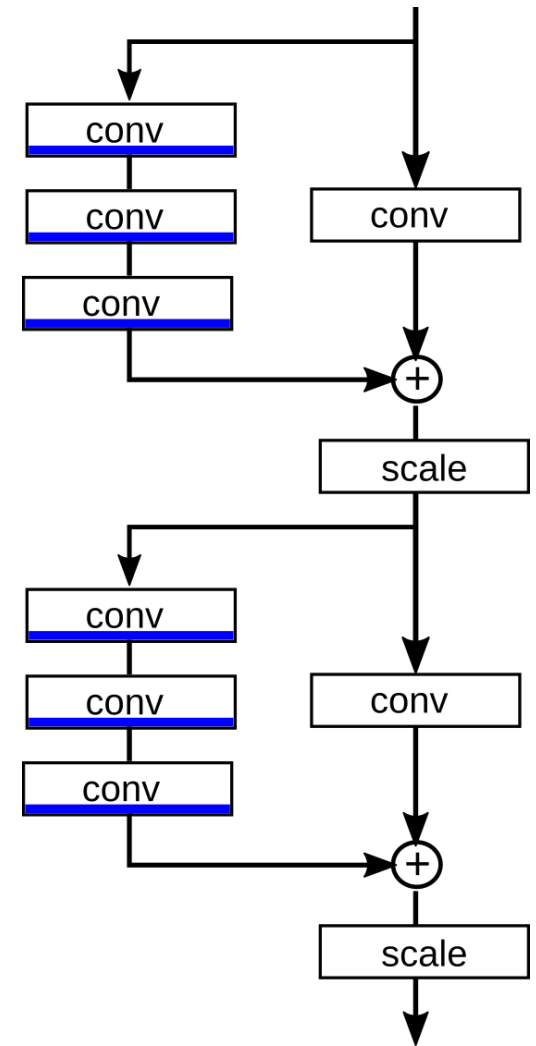
Let us have a different view on residual networks. We have two linked networks:

- ◆ a “highway” network with few layers,
- ◆ a very deep network which adds “corrections” the former.

This improves trainability of the network. It becomes possible to train networks with 100 and more layers.

Attempts for theoretical explanation:

- ◆ analyse the gradients statistic. Resnets have rather uniform distribution of gradients,
- ◆ interpret resnets as compositions of “near to identity” mappings and model them by kernels.



Transfer learning: pre-training + fine-tuning

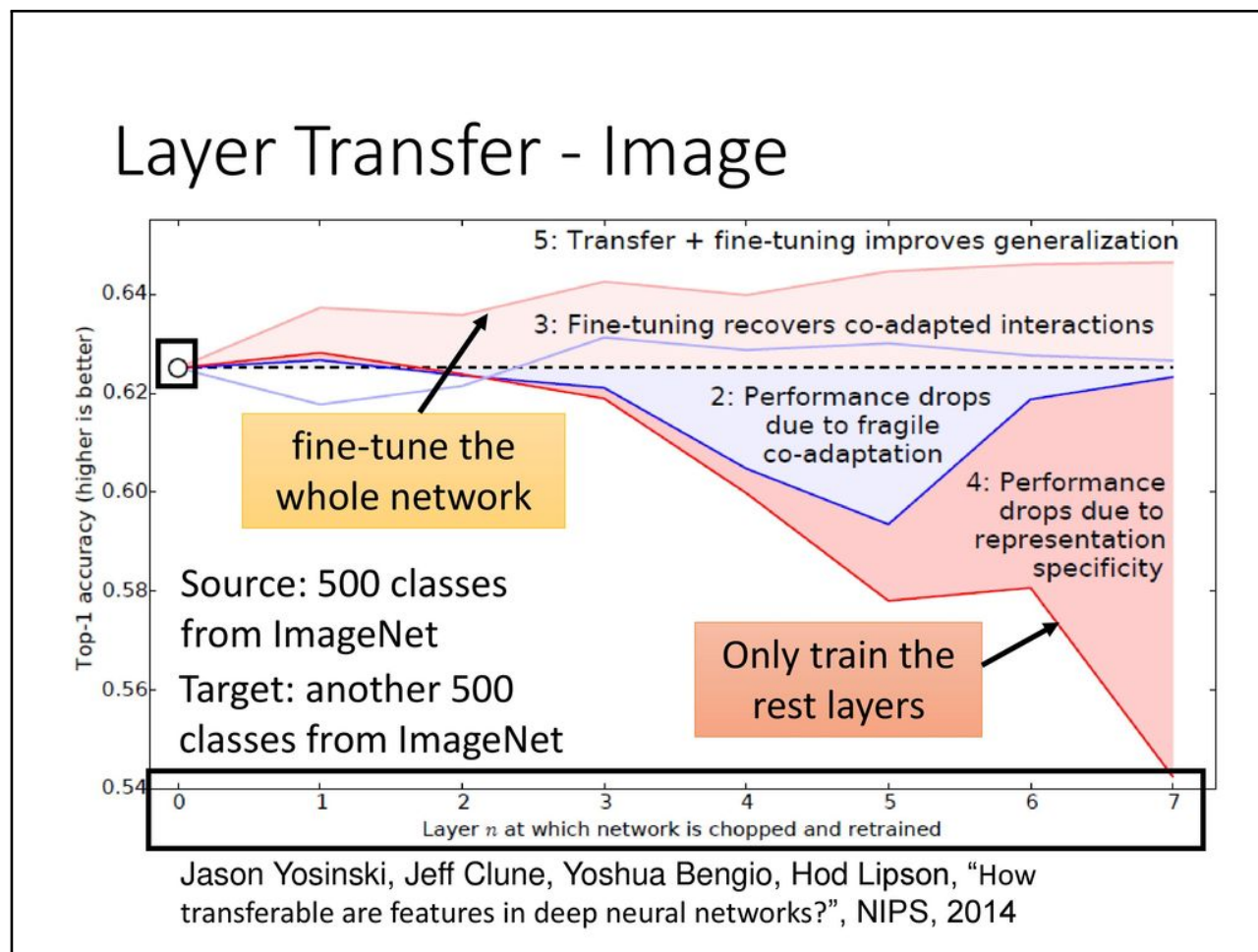
- ◆ You want to train a predictor for a complex recognition task, but suffer from lack of training data.
- ◆ A predictor for a different task has been successfully trained on a large dataset.
- ◆ The domains of the two tasks are similar.

We can use the following approach

- ◆ Use the first layers of the network that implements the predictor for the other task.
- ◆ Add your layers on top
- ◆ Learn the network on your data, if necessary apply early stopping to prevent overfitting.
This can be done in two ways
 - (1) freeze the parameters of the transferred layers
 - (2) fine-tuning: learn parameters of all layers

Transfer Learning: pre-training & fine-tuning

Example 2 (Yosinski et al., NIPS 2014). Randomly split the 1000 Image-Net classes into two groups with 500 classes: datasets A and B . Learn BnB , BnB^+ , AnB and AnB^+ networks. Here: letters indicate the task of the pre-trained/transfer network, n is the layer number and $+$ indicate the fine-tuning variant.



blue: BnB , BnB^+ red: AnB , AnB^+