Deep Learning (BEV033DLE)
Lecture 6 Weight initialisation, batch normalisation, data augmentation

Czech Technical University in Prague

- Weight initialisation
- Batch normalisation
- Data augmentation
- Transfer learning

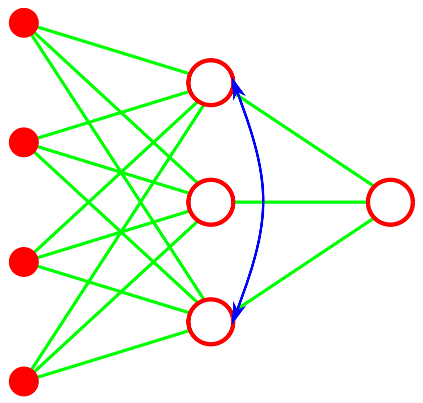
Weight initialisation

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

Side step: symmetries and gradients:

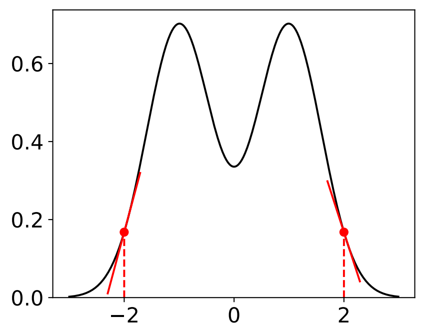
Consider a scalar function $f(w)$ that is invariant to the linear mapping $B\colon \mathbb{R}^n \to \mathbb{R}^n$, i.e. $f(Bw) = f(w)$. Its gradient ∇f has the property

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



which follows from

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



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

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

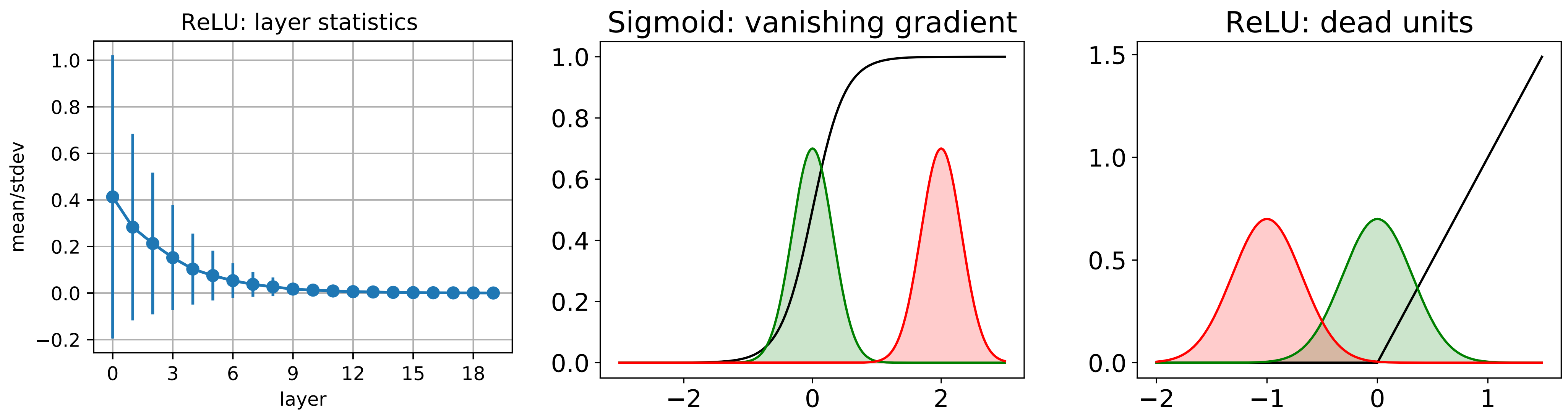
The new point w_1 will be again invariant, i.e. $Bw_1 = w_1$. We need to break the symmetry!

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: statistics over the layers for 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

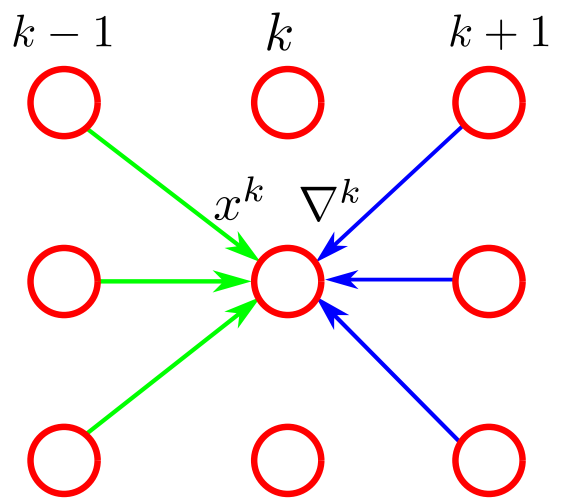
(3) **Proper initialisation:** Initialise weights/biases so that each neuron has activation statistic (over the dataset) with certain mean and variance.

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 $y = w^T x$, $x \in \mathbb{R}^n$. We have $\mathbb{V}[y] \approx n\mathbb{V}[w]\mathbb{V}[x]$. Denote variance of weights in layer k by v_k, neuron outputs by x^k, gradients by ∇^k and number of neurons by n_k.

- forward: $\mathbb{V}[x^k] = n_{k-1} v_k \mathbb{V}[x^{k-1}]$
  We want $\mathbb{V}[x^k] \approx \mathbb{V}[x^{k-1}]$, i.e. $n_{k-1} v_k = 1$.
- backward: $\mathbb{V}[\nabla^k] = n_{k+1} v_{k+1} \mathbb{V}[\nabla^{k+1}]$
  We want $\mathbb{V}[\nabla^k] \approx \mathbb{V}[\nabla^{k+1}]$, i.e. $n_k v_k = 1$
- Compromise: Set $v_k = \frac{2}{n_{k-1}+n_k}$. Assuming that the inputs x^0 have zero mean and unit variance, initialise the weights randomly by $w^k_{ij} \sim \mathcal{N}(0, \sqrt{v_k})$.



Similar considerations for ReLU activation lead to a different scheme (He et al., 2015)

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

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

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

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

- 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

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

Data augmentation

Goals of data augmentation:

- Artificially enlarge the training set – an attempt to bound the generalisation error (i.e. prevent overfitting).
- Enforce invariance of the predictor w.r.t. certain transformations of the input space.

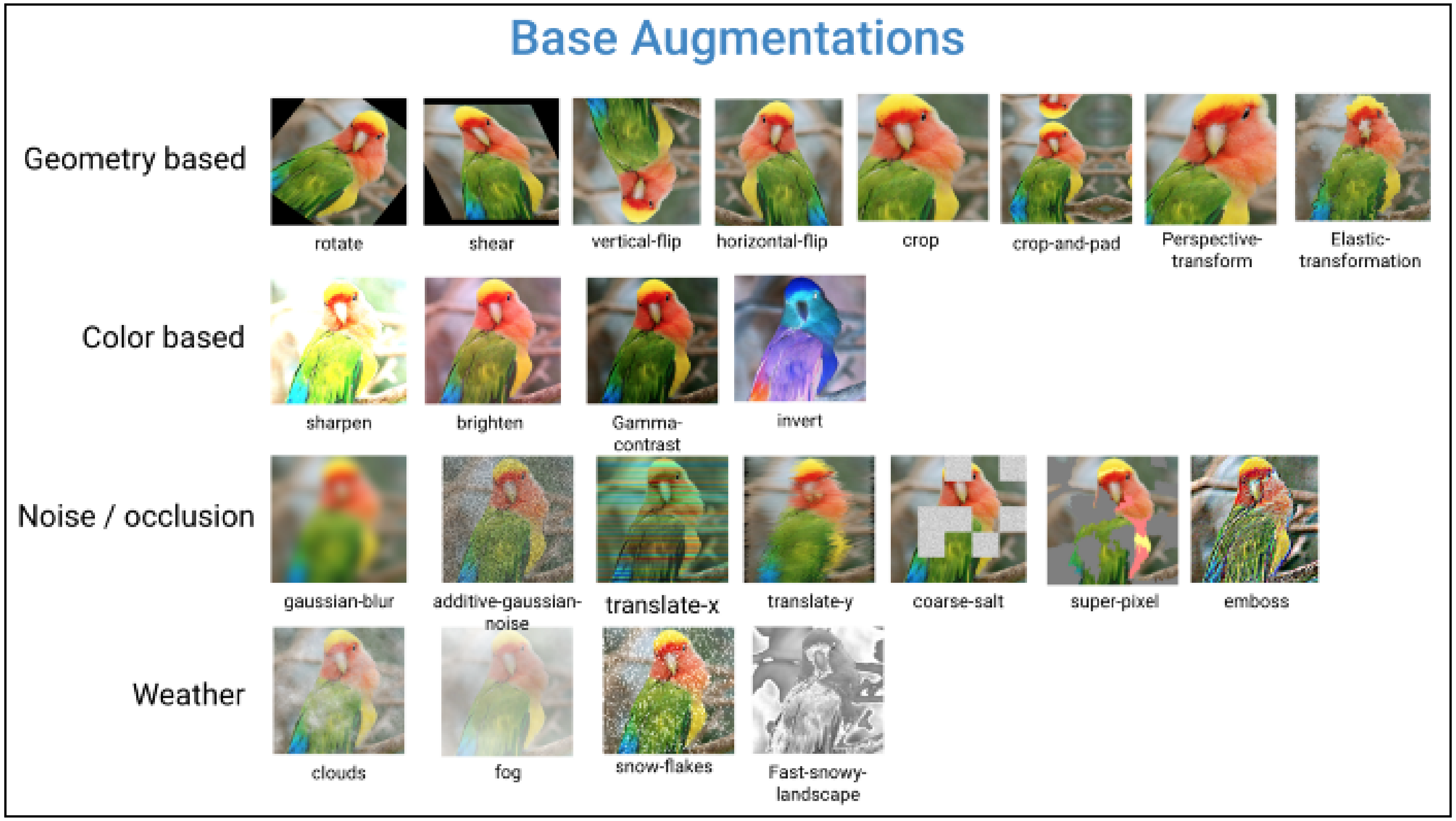
Technically: online augmentation generates new data on the fly, whereas offline augmentation stores augmented datasets.

We discuss it here in context of image processing (classification, segmentation . . .)

(Image) data augmentation: Create new images from a single training image

- geometric transformations: flip, crop, rotate, non-linear transformations,. . .
- photometric transformations: color space transformations, histogram changes,. . .
- kernel transforms: sharpening, blurring,. . .
- noise: pixel-wise independent noise, jitter, random erasing,. . .

Data augmentation



Available libraries & methods: Augmentor, Albumentations, DeepAugment, GAN based style transfer, ...

Transfer Learning: pre-training & fine-tuning

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

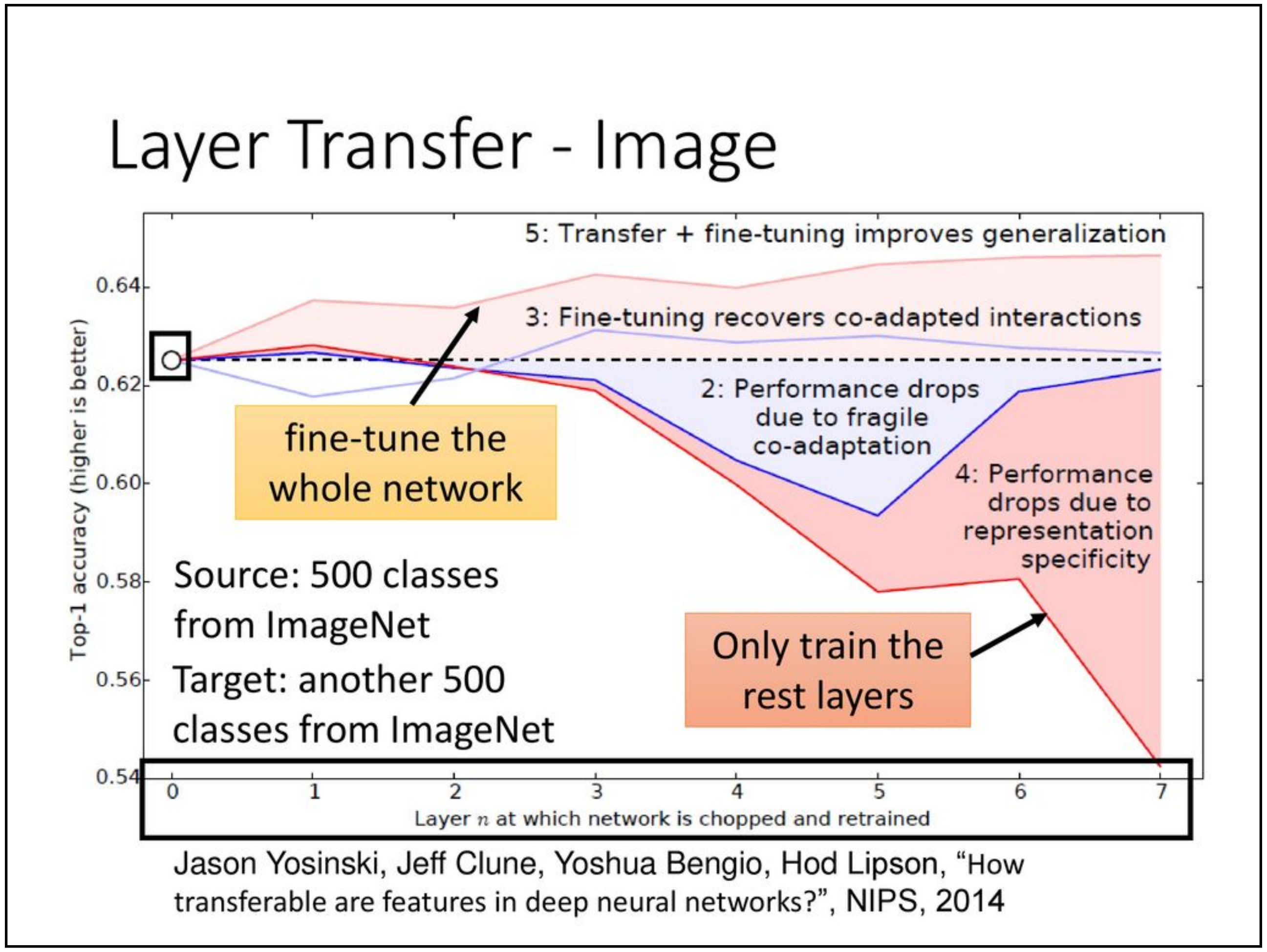
- Use the first layers of the network that implements the predictor for the other task.
- Add your layers on top
- Learn the network on your data, if necessary apply early stopping to prevent overfitting. This can be done in two ways

  (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^+