Deep Learning (BEV033DLE)
Lecture 8 Training Neural Networks 1.

Czech Technical University in Prague

- Data augmentation
- Weight initialisation
- Batch normalisation

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 . . .)

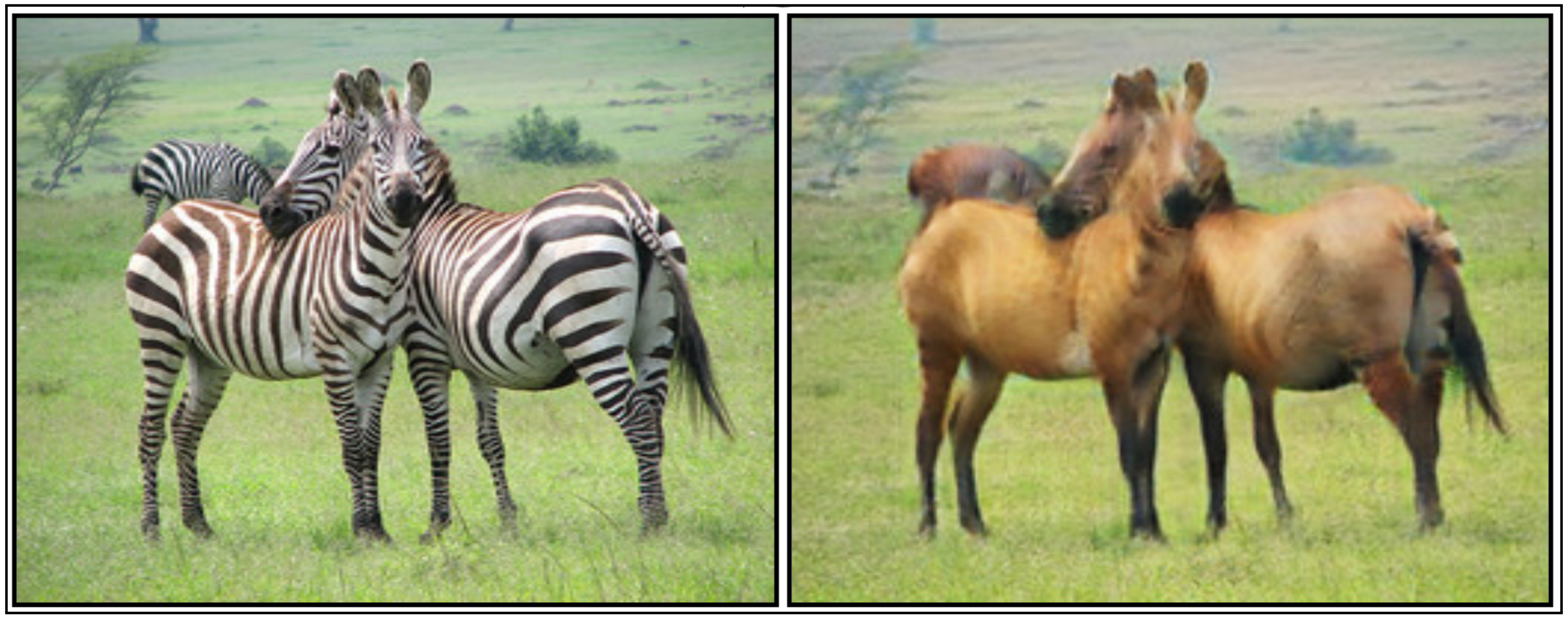
(1) Image data augmentation: Create a new image from a single training image

- geometric transformations: flip, crop, rotate, nonlinear transformations,. . .
- photometric transformations: color space transformations, histogram changes,. . .
- kernel transforms: sharpening, blurring,. . .
- noise: pixelwise independent noise, jitter, random erasing,. . .

Data augmentation

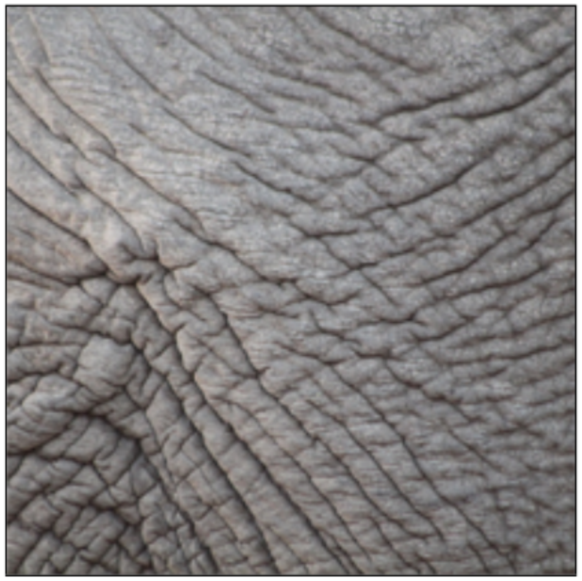
(2) NN based augmentation:

- Apply distortions and noise on the level of intermediate NN features
- NNs for generating new images from image pairs
- use VAEs and its conditional versions: a VAE learns to map a noise space onto an image domain.
- style transfer & cycle GANs: A cycle GAN maps image domains onto each other, without the need to have paired training data.



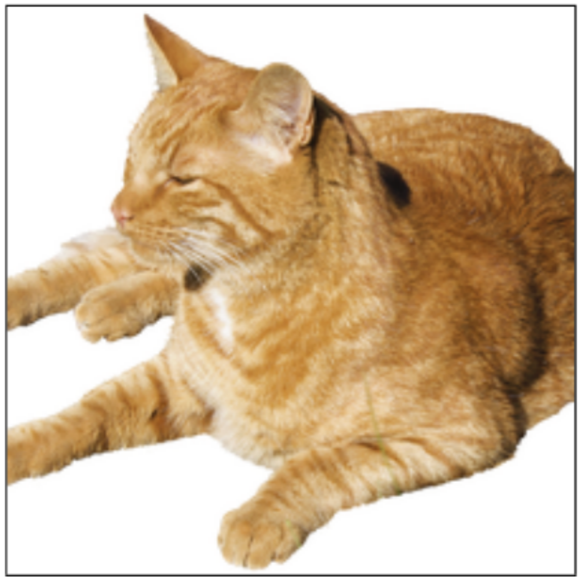
Data augmentation

Geirhos et al., ImageNet-trained CNNs are biased towards texture; Increasing shape bias improves accuracy and robustness, ICLR 2019



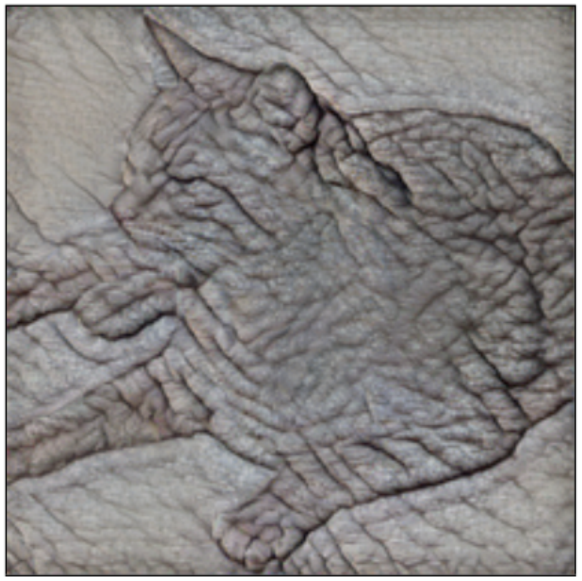
(a) Texture image

81.4%	**Indian elephant**
10.3%	indri
8.2%	black swan



(b) Content image

71.1%	**tabby cat**
17.3%	grey fox
3.3%	Siamese cat



(c) Texture-shape cue conflict

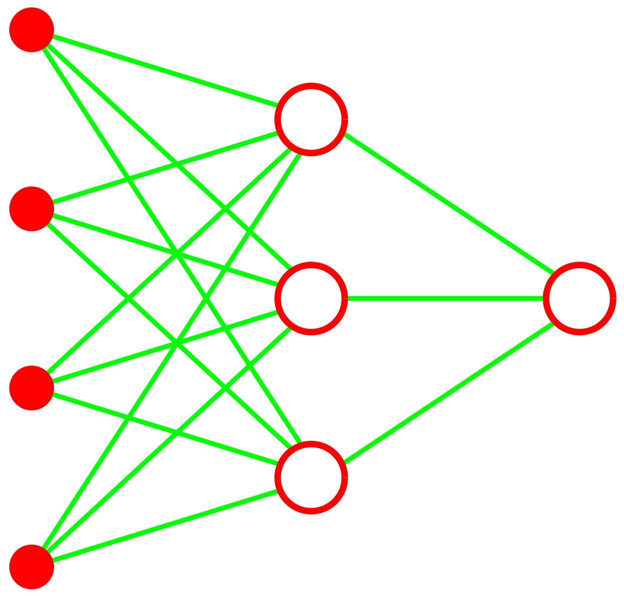
63.9%	**Indian elephant**
26.4%	indri
9.6%	black swan

Weight initialisation

Question: How to initialise the parameters, i.e. weights and biases, of a network?

(1) Initialising all weights and biases with zero is bad because networks are invariant w.r.t. permutations of neurons.

- This network is invariant w.r.t. permuting the neurons in the hidden layer and the weights of the output neuron.
- If we start training from zero weights and biases, will keep having identical weights and biases.
- The same holds for the weights of the output neuron – they will keep being equal.



Explanation: Let $f\colon \mathbb{R}^n \to \mathbb{R}$ be invariant w.r.t. a linear operator B, i.e. $f(Bw) = f(w)$ for all w. The we can compute its gradient ∇f in Bw from its gradient in w:

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

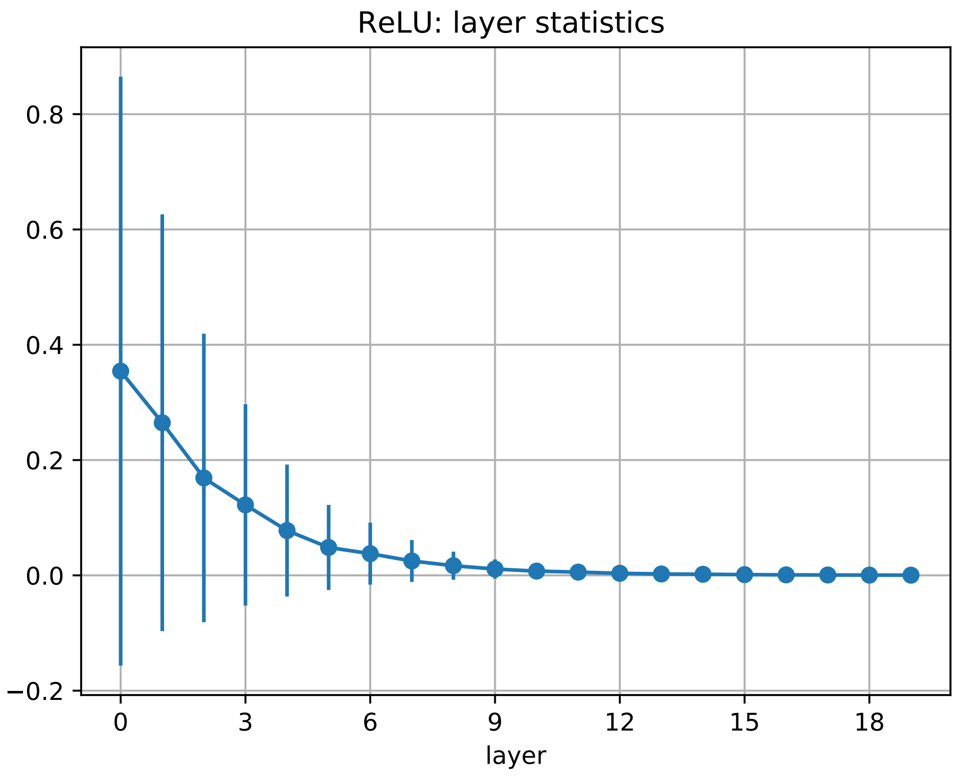
Notice also, that we have $B^{-T} = B$ if B is a permutation.

Weight initialisation

If SGD is started from an invariant point $w_0 = Bw_0$, we have

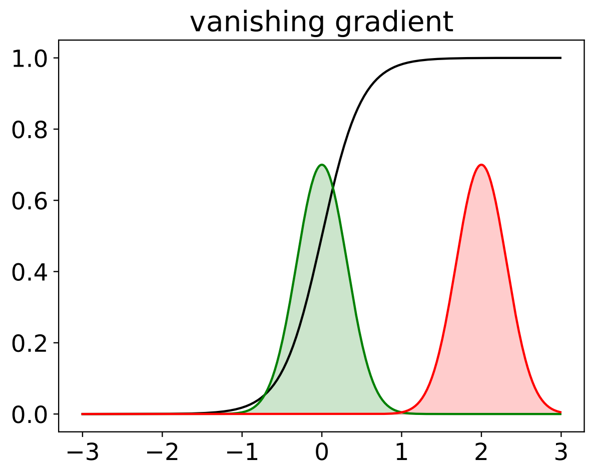
$$B\big[w_0 + \alpha \nabla f(w_0)\big] = w_0 + \alpha \nabla f(Bw_0) = w_0 + \alpha \nabla f(w_0)$$

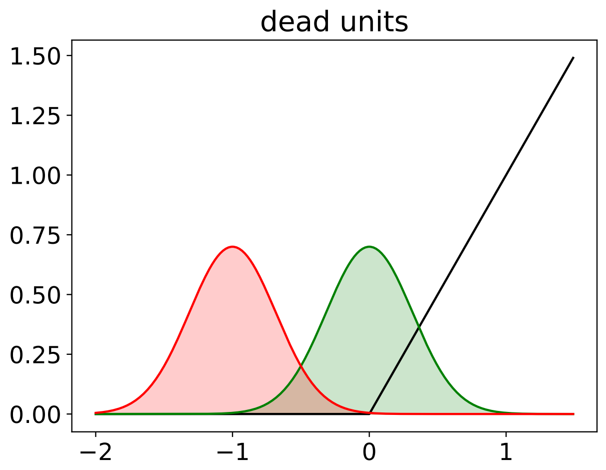
(2) Unfortunately, initialising all weights and biases of a deep network randomly from a uniform distribution (or a normal distribution) is not a good idea either.



Weight initialisation

This can lead to vanishing/exploding gradients and "dead units" during learning





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

This can be, in principle, achieved by the following "forward initialisation pass"

. . .

Layer k:

- randomly pre-initialise the weights by $w_{ij}^k \sim \mathcal{N}(0,1)$
- compute statistic for neuron activations
- rescale weights and set biases so as to achieve the desired activation statistic

. . .

Weight initialisation

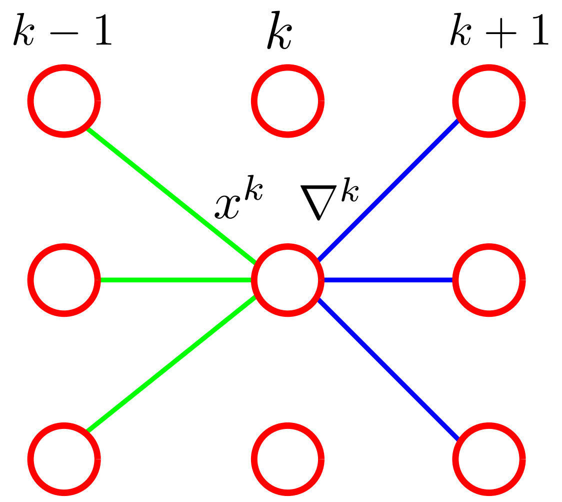
(4) (Glorot & Bengio, 2010) Analyse variance of neuron outputs and backprop gradients under the following simplifying assumptions

- 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 minibatch,
- 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

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

Reparametrisation and gradient preconditioning

Let $f(x)$ be a function on $\mathbb{R}^n$, which we minimise by gradient descent

$$x_{k+1} = x_k + \alpha \nabla f(x_k).$$

Using new coordinates $x = H(y)$ and defining $g = f \circ H$, we can use instead gradient descent of for g

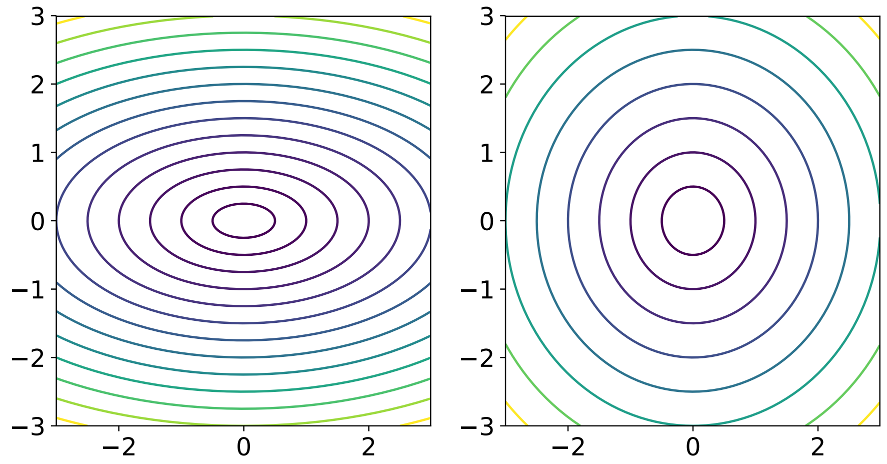
$$y_{k+1} = y_k + \alpha \nabla g(x_k),$$

where the gradient of g is obtained form the gradient of f by

$$\langle \nabla g(y), dy \rangle = \big\langle \nabla f(H^{-1}y), J_y^{-1} dx \big\rangle = \big\langle J_y^{-T} \nabla f(H^{-1}y), dx \big\rangle, \tag{1}$$

where J_y denotes the Jacobian of the mapping G in the point y.

Question: which of the two versions of the gradient descent can be made converging faster?



Reparametrisation and gradient preconditioning

(Static) preconditioning of gradient descent

$$x_{k+1} = x_k + \alpha P \nabla f(x_k),$$

where P is a positive definite matrix.

Another view: Newton method

$$x_{k+1} = x_k + \alpha \big[Hf(x_k)\big]^{-1} \nabla f(x_k),$$

where $Hf(x_k)$ is the Hessian of f in x_k. Now, approximate the Hessian by a constant matrix $P^{-1} \approx Hf(x^*)$.