Training Neural Networks II

Milan Straka

March 4, 2024

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Putting It All Together

Let us have a dataset with training, validation, and test sets, each containing examples $(\boldsymbol{x}, y)$. Depending on y, consider one of the following output activation functions:

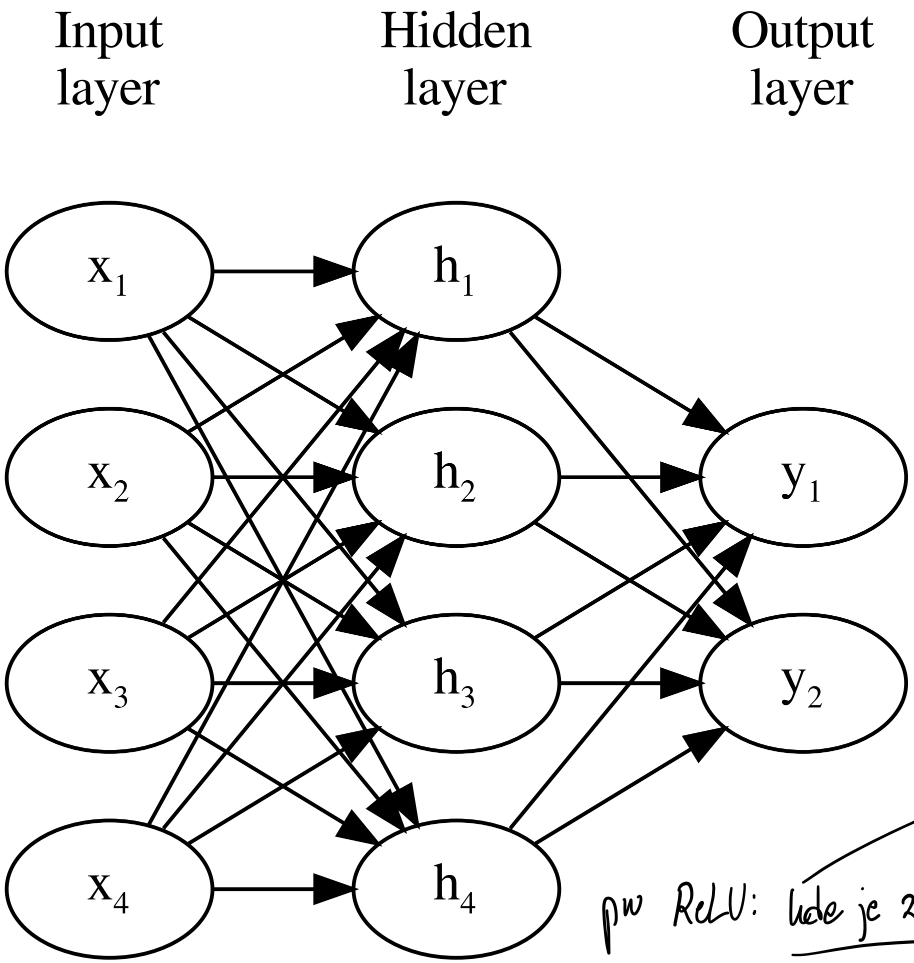
vstup výstup

$$\begin{cases} \text{none} & \text{if } y \in \mathbb{R} \text{ and we assume variance is constant everywhere,} \\ \sigma & \text{if } y \text{ is a probability of a binary outcome,} \\ \operatorname{softmax} & \text{if } y \text{ is a gold class index out of } K \text{ classes (or a full distribution).} \end{cases}$$

If $\boldsymbol{x} \in \mathbb{R}^D$, we can use a neural network with an input layer of size D, some number of hidden layers with nonlinear activations, and an output layer of size O (either 1 or the number of classes K) with the mentioned output function.

There are of course many functions, which could be used as output activations instead of σ *and* softmax*; however,* σ *and* softmax *are almost universally used. One of the reason is that they can be derived using the maximum-entropy principle from a set of conditions, see the Machine Learning for Greenhorns (NPFL129) lecture 5 slides. Additionally, they are the inverses of canonical link functions of the Bernoulli and categorical distributions, respectively.*

Putting It All Together – Single-Hidden-Layer MLP



We have

$$h_i = f^{(1)}\left(\sum_j x_j W^{(1)}_{j,i} + b^{(1)}_i\right)$$

where

- $\boldsymbol{W}^{(1)} \in \mathbb{R}^{D\times H}$ is a matrix of **weights**,
- $\boldsymbol{b}^{(1)} \in \mathbb{R}^{H}$ is a vector of **biases**,
- $f^{(1)}$ is an activation function.

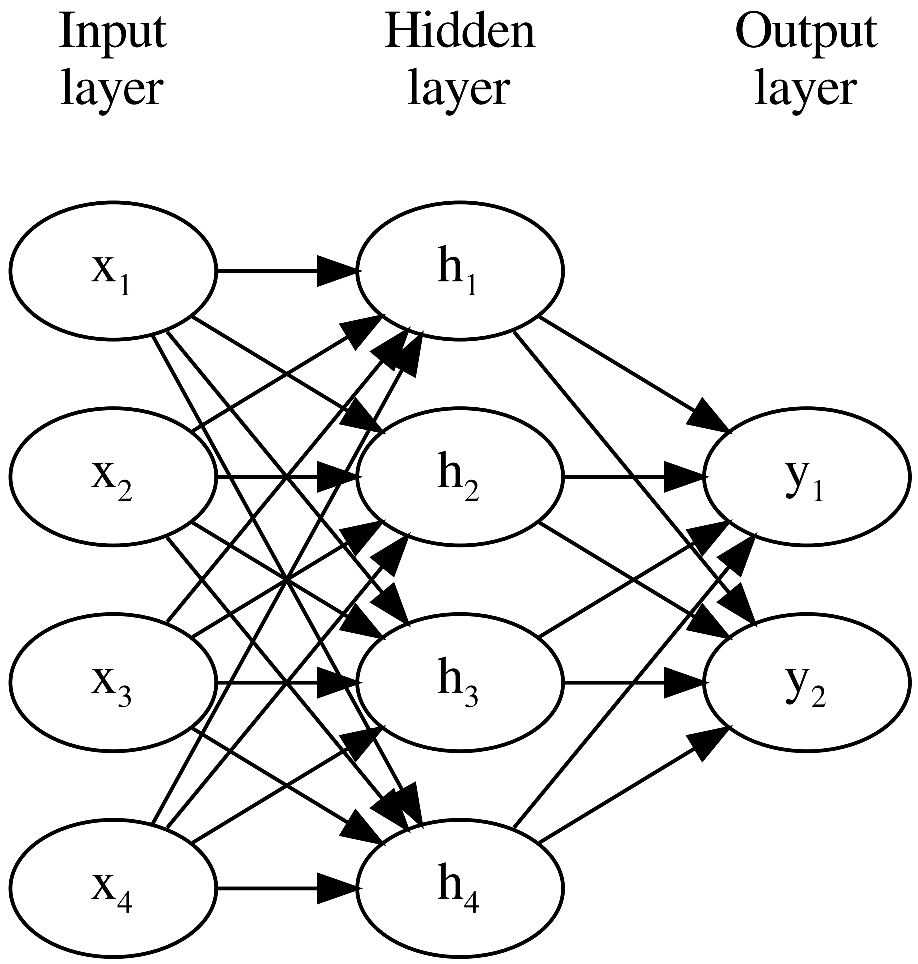
The weight matrix is also called a **kernel**.

The biases define general behaviour in case of zero/very small input.

pw ReLU: kde je zlom?

Transformations of type $\boldsymbol{x}^T\boldsymbol{W}^{(1)} + \boldsymbol{b}$ are called **affine** instead of *linear*.

Putting It All Together – Single-Hidden-Layer MLP



Similarly

$$o_i = f^{(2)}\left(\sum_j h_j W^{(2)}_{j,i} + b^{(2)}_i\right)$$

with

- $\boldsymbol{W}^{(2)} \in \mathbb{R}^{H\times O}$ another matrix of weights,
- $\boldsymbol{b}^{(2)} \in \mathbb{R}^{O}$ another vector of biases,
- $f^{(2)}$ being an output activation function.

Putting It All Together – Parameters and Training

Altogether, the $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(1)}$, and $\boldsymbol{b}^{(2)}$ form the **parameters** of the model, which we denote as a vector $\boldsymbol\theta$ in the model description and machine learning algorithms.

In our case, the parameters have a total size of $D \times H + H \times O + H + O$.

To train the network, we repeatedly sample m training examples and perform an SGD (or any of its adaptive variants), updating the parameters to minimize the loss derived by ~~MSE~~ MLE $E(\boldsymbol\theta) = \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim\hat{p}_\text{data}} L(f(\boldsymbol{x};\boldsymbol\theta), y)$:

$$\theta_i \leftarrow \theta_i - \alpha\frac{\partial E(\boldsymbol\theta)}{\partial\theta_i},\ \ \text{or in vector notation,}\ \ \boldsymbol\theta \leftarrow \boldsymbol\theta - \alpha\nabla_{\boldsymbol\theta} E(\boldsymbol\theta).$$

We set the hyperparameters (size of the hidden layer, hidden layer activation function, learning rate, …) using performance on the validation set and evaluate generalization error on the test set.

Putting It All Together – Batches

- We always process data in **batches**, i.e., matrices whose rows are the batch examples.
- We represent the network in a vectorized way (tensorized would be more accurate).

Instead of $H_{b,i} = f^{(1)}\left(\sum_j X_{b,j}W_{j,i}^{(1)} + b_i^{(1)}\right)$, we compute

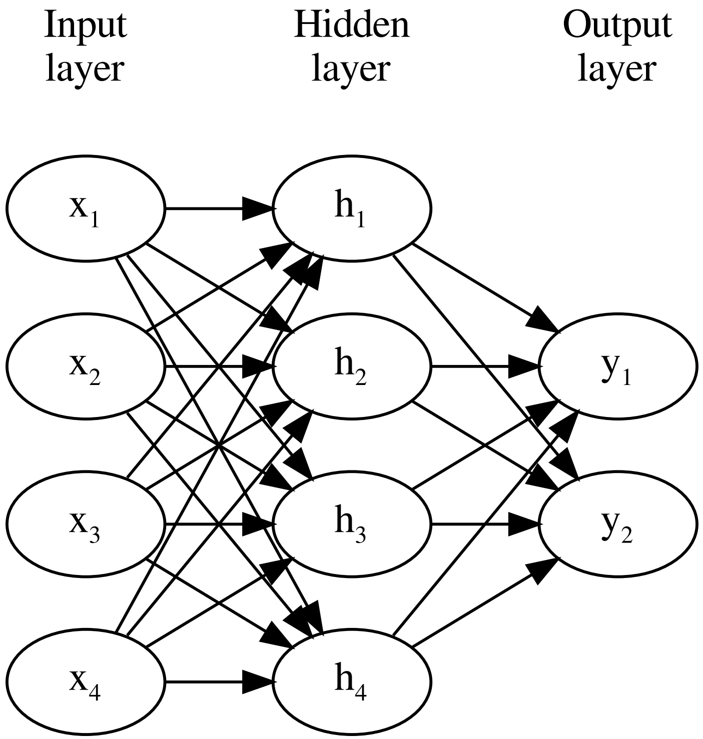
$$\boldsymbol{H} = f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right),$$
$$\boldsymbol{O} = f^{(2)}\left(\boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right) = f^{(2)}\left(f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right).$$

The derivatives

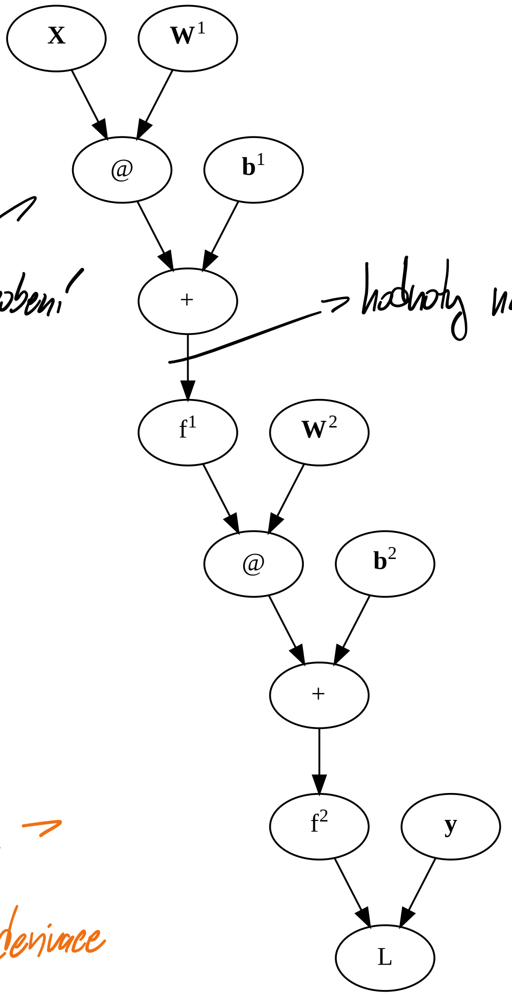
$$\frac{\partial f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)}{\partial \boldsymbol{X}}, \frac{\partial f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)}{\partial \boldsymbol{W}^{(1)}}, \ldots$$

are then batches of matrices (called **Jacobians**) or even higher-dimensional tensors.

Putting It All Together – Computation Graph



$\rightarrow$



maticové násobení

hodnoty na skryté vrstvy

takhle spočítám ty derivace

Putting It All Together – Designing and Training Neural

Designing and training a neural network is not a one-shot action, but instead an iterative procedure.

- When choosing hyperparameters, it is important to verify that the model does not underfit and does not overfit.
- Underfitting can be checked by trying increasing model capacity or training longer, and observing whether the training performance increases.
- Overfitting can be tested by observing train/dev difference, or by trying stronger regularization and observing whether the development performance improves.

Regarding hyperparameters:

- We need to set the number of training epochs so that development performance stops increasing during training (usually later than when the training performance plateaus).
- Generally, we want to use large enough batch size, but such a one which does not slow us down too much (GPUs sometimes allow larger batches without slowing down training). However, because larger batch size implies less noise in the gradient, small batch size sometimes work as regularization (especially for vanilla SGD algorithm).

High Level Overview

	Classical ('90s)	mít jenom více skrytých vrstev moc nepomáhá Deep Learning
Architecture	$\vdots\ \vdots\ \vdots$	$\vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots\ \vdots$ CNN, RNN, Transformer, VAE, GAN, …
Activation func.	$\tanh, \sigma$	$\tanh$, ReLU, LReLU, GELU, Swish (SiLU), SwiGLU, …
Output function	none, σ	none, σ, $\operatorname{softmax}$
Loss function	MSE	NLL (or cross-entropy or KL-divergence)
Optimization	SGD, momentum	SGD ($+$ momentum), RMSProp, Adam, SGDW, AdamW, …
Regularization	L^2, L^1	L^2, Dropout, Label smoothing, BatchNorm, LayerNorm, MixUp, WeightStandardization, …

Metrics and Losses

During training and evaluation, we use two kinds of error functions:

- **loss** is a *differentiable* function used during training,
  - NLL, MSE, Huber loss, Hinge, …
- **metric** is any (and very often non-differentiable) function used during evaluation,
  - any loss, accuracy, F-score, BLEU, …
  - possibly even human evaluation.

In Keras, the losses and metrics are available in `keras.losses` and `keras.metrics`.

Keras Losses

The `keras.losses` offer two sets of APIs. The high-level API ones are loss classes like

```
keras.losses.MeanSquaredError(
    reduction="sum_over_batch_size", name="mean_squared_error"
)
```

→ tohle je obyč „avg"

The created objects are subclasses of `kerass.losses.Loss` and can be always called with three arguments:

→ specifické váhy pro konkrétní samples z batche

```
__call__(y_true, y_pred, sample_weight=None)
```

which returns the loss of the given data, *reduced* using the specified reduction. If `sample_weight` is given, it is used to weight (multiply) the individual batch example losses before reduction.

- `reduction="sum_over_batch_size"`
- `reduction="sum"`
- `reduction=None`

Keras Cross-entropy Losses

The cross-entropy losses need to specify also the distribution in question:

- `keras.losses.BinaryCrossentropy`: the gold and predicted distributions are Bernoulli distributions (i.e., a single probability);
- `keras.losses.CategoricalCrossentropy`: the gold and predicted distributions are categorical distributions;
- `keras.losses.SparseCategoricalCrossentropy`: a special case, where the gold distribution is one-hot distribution (i.e., a single correct class), which is represented as the gold *class index*; therefore, it has one less dimension than the predicted distribution.

These losses expect probabilities on input, but offer `from_logits` argument, which can be used to indicate that logits are used instead of probabilities, which is more numerically stable.

Functional Losses API

In addition to the loss objects, `keras.losses` offers methods like `keras.losses.mean_squared_error`, which process two arguments `y_true` and `y_pred` and do not reduce the batch example losses.

Keras Metrics

There are two important differences between metrics and losses.

1. metrics may be non-differentiable;
2. metrics **aggregate** results over multiple batches.

The metric objects are subclasses of `keras.metrics.Metric` and offer the following methods:

- `update_state(y_true, y_pred, sample_weight=None)` updates the value of the metric and stores it;
- `result()` returns the current value of the metric;
- `reset_states()` clears the stored state of the metric.

The most common pattern is using the provided method

```
__call__(y_true, y_pred, sample_weight=None)
```

which is a combination of `update_state` followed by a `result()`.

Keras Metrics

Apart from analogues of the losses

- `keras.metrics.MeanSquaredError`
- `keras.metrics.BinaryCrossentropy`
- `keras.metrics.CategoricalCrossentropy`
- `keras.metrics.SparseCategoricalCrossentropy`

vyplatí se tedy vybrat vhodnou metriku vzhledem k targetům

the `keras.metrics` module provides

→ např. pokud y_pred = 0,7 a y_true = 1, tak Accuracy dá 0, BinaryAccuracy dá 1...

- `keras.metrics.Mean` computing averaged mean;
- `keras.metrics.Accuracy` returning accuracy, which is an average number of examples where the prediction is equal to the gold value;
- `keras.metrics.BinaryAccuracy` returning accuracy of predicting a Bernoulli distribution (the gold value is 0/1, the prediction is a probability);
- `keras.metrics.CategoricalAccuracy` returning accuracy of predicting a Categorical distribution (the argmaxes of gold and predicted distributions are equal);
- `keras.metrics.SparseCategoricalAccuracy` is again a special case of `CategoricalAccuracy`, where the gold distribution is represented as the gold class *index*.

Derivative of MSE Loss

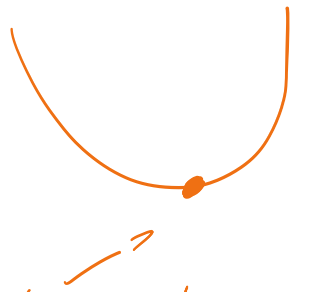
Given the MSE loss of

$$L = \big(f(\boldsymbol{x};\boldsymbol{\theta}) - y\big)^2,$$

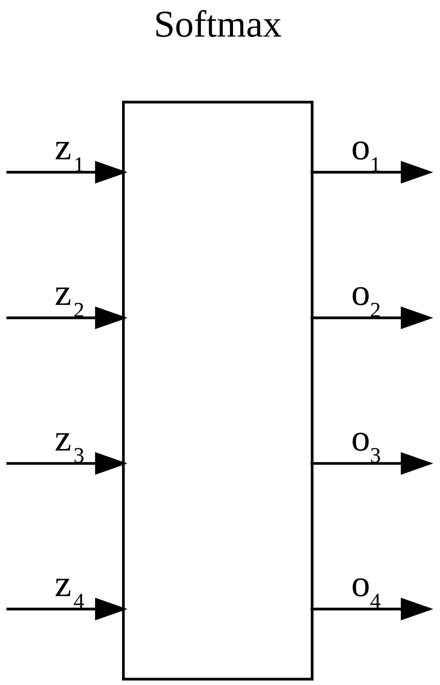
the derivative with respect to the model output is simply:

$$\frac{\partial L}{\partial f(\boldsymbol{x};\boldsymbol{\theta})} = 2\big(f(\boldsymbol{x};\boldsymbol{\theta}) - y\big).$$

$$\sim \hat{y} - y =$$



Derivative of Softmax MLE Loss



Let us have a softmax output layer with

$$o_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

→ normalizace, aby $\in \langle 0, 1 \rangle$

Derivative of Softmax MLE Loss

Consider now the MLE estimation. The loss for gold class index $gold$ is then

$$L(\operatorname{softmax}(\boldsymbol z), gold) = -\log o_{gold}.$$

The derivation of the loss with respect to $\boldsymbol z$ is then

$$\begin{aligned}
\frac{\partial L}{\partial z_i} = \frac{\partial}{\partial z_i}\left[-\log \frac{e^{z_{gold}}}{\sum_j e^{z_j}}\right] &= -\frac{\partial z_{gold}}{\partial z_i} + \frac{\partial \log(\sum_j e^{z_j})}{\partial z_i} \\
&= -[gold = i] + \boxed{\frac{1}{\sum_j e^{z_j}} e^{z_i}} \\
&= -[gold = i] + o_i .
\end{aligned}$$

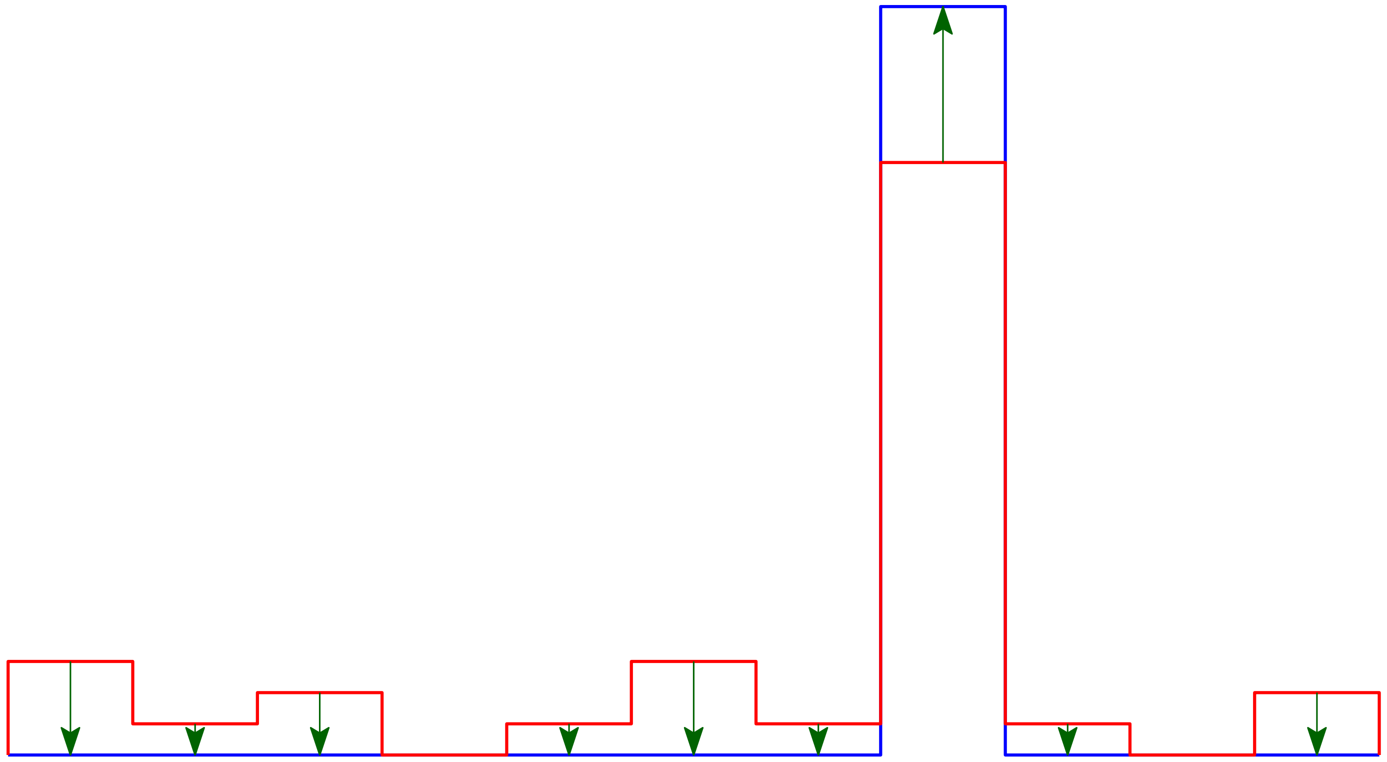
$\rightarrow$ tohle je softmax

„output – co jsme měli predikovat"

$\rightarrow$ tedy pokud jsem predikoval správně dostanu loss 0. ✓

Therefore, $\frac{\partial L}{\partial \boldsymbol z} = \boldsymbol o - \boldsymbol 1_{gold}$, where $\boldsymbol 1_{gold}$ is the one-hot encoding (a vector with 1 at the index $gold$ and 0 everywhere else).

Derivative of Softmax MLE Loss



Gold distribution

Model distribution

Loss derivative with respect to the softmax inputs.

Derivative of Softmax and Sigmoid MLE Losses

In the previous case, the gold distribution was *sparse*, with only one target probability being 1.

In the case of general gold distribution $\boldsymbol{g}$, we have

$$L(\mathrm{softmax}(\boldsymbol{z}), \boldsymbol{g}) = -\sum\nolimits_i g_i \log o_i.$$

Repeating the previous procedure for each target probability, we obtain

$$\frac{\partial L}{\partial \boldsymbol{z}} = \boldsymbol{o} - \boldsymbol{g}.$$

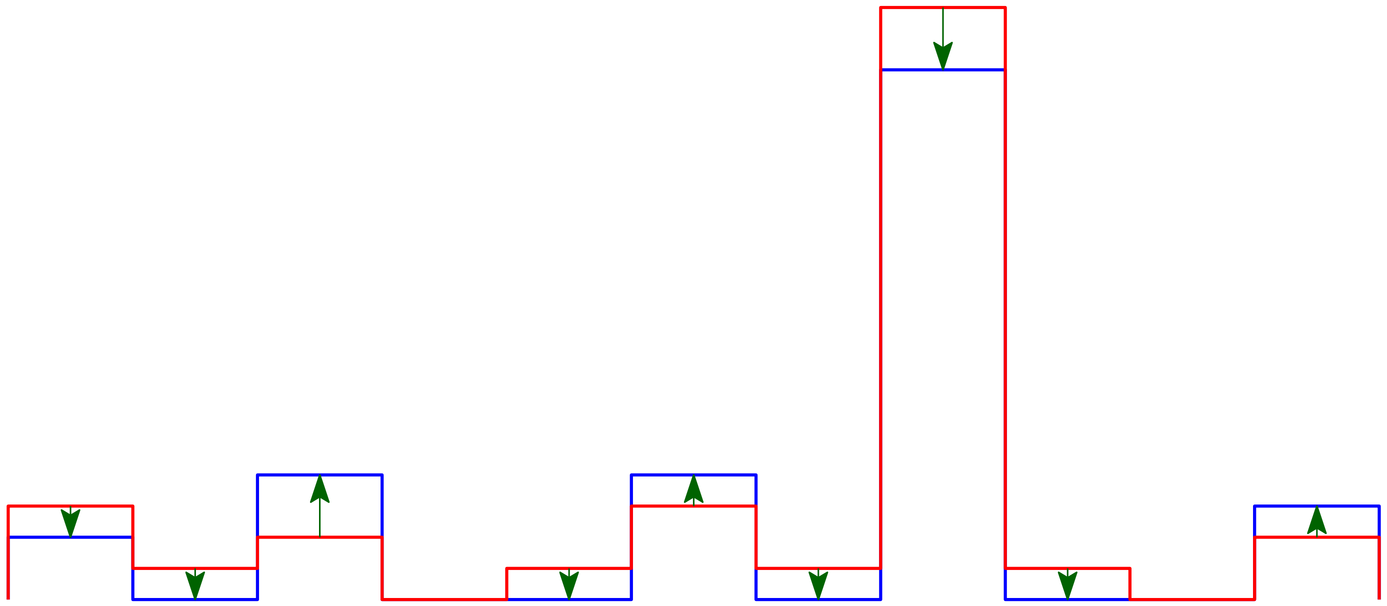
Sigmoid

Analogously, for $o = \sigma(z)$ we get $\frac{\partial L}{\partial z} = o - g$, where g is the target gold probability.

The result follows automatically from the fact that σ can be computed using $\mathrm{softmax}$ as

$$\mathrm{softmax}\left(\begin{bmatrix}0 & x\end{bmatrix}\right)_1 = \frac{e^x}{e^x + e^0} = \frac{1}{1 + e^{-x}} = \sigma(x).$$

Derivative of Softmax MLE Loss



Gold distribution

Model distribution

Loss derivative with respect to the softmax inputs.

Regularization

As already mentioned, regularization is any change in the machine learning algorithm that is designed to reduce generalization error but not necessarily its training error.

Regularization is usually needed only if training error and generalization error are different. That is often not the case if we process each training example only once. Generally the more training data, the better generalization performance without any explicit regularization.

- Early stopping easy:
- L^2, L^1 regularization
- Dataset augmentation
- Ensembling
- Dropout
- Label smoothing

Kdybych měl spoustu tr. dat, nemusel bych regularizovat.
– to ale standardní nemám...

Stará technika, co se ještě používá

Regularization – Early Stopping

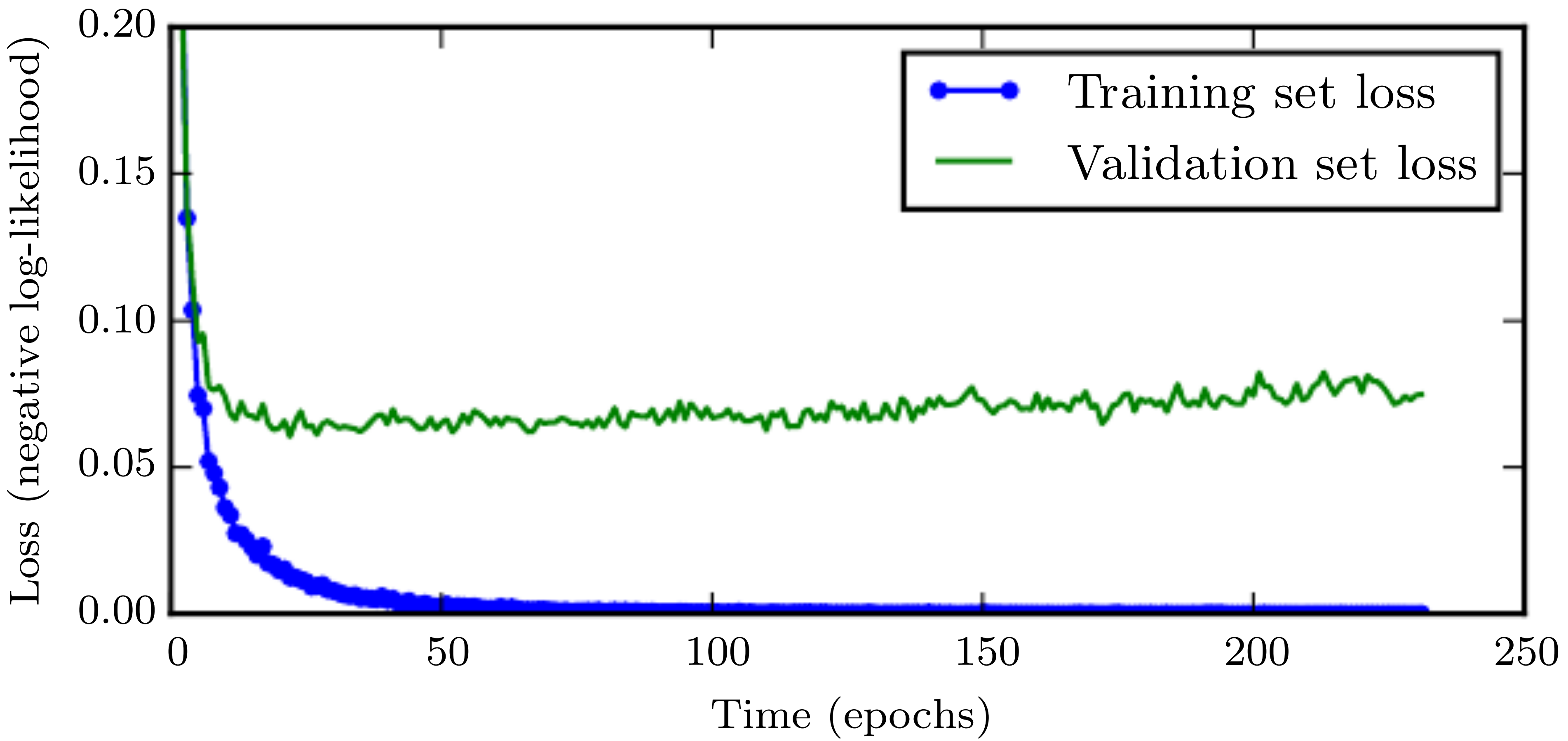


Figure 7.3 of "Deep Learning" book, https://www.deeplearningbook.org

L2 Regularization

L^2-regularization is one of the oldest regularization techniques, which tries to prefer "simpler" models by endorsing models with **smaller weights**.

Concretely, **L^2-regularization** (also called **Tikhonov regularization** or **weight decay**) penalizes models with large weights by utilizing the following error function:

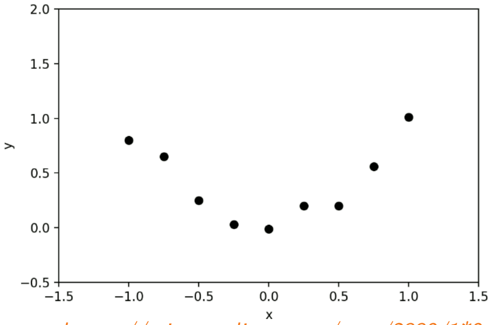
$$\tilde{E}(\boldsymbol{\theta}; \mathbb{X}) = E(\boldsymbol{\theta}; \mathbb{X}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2$$

for a suitable (usually very small) λ.

Note that the L^2-regularization is usually not applied to the *bias*, only to the "proper" weights, because we cannot really overfit via the bias.

L2 Regularization

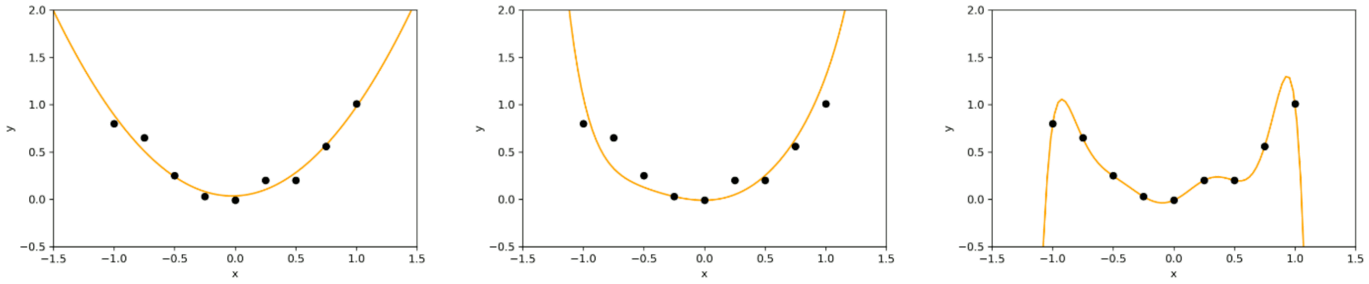
One way to look at L^2-regularization is that it promotes smaller changes of the model (the Jacobian of a single layer with respect to the inputs depends on the weight matrix, because $\frac{\partial \boldsymbol{x}^T \boldsymbol{W}}{\partial \boldsymbol{x}} = \boldsymbol{W}$).



https://miro.medium.com/max/2880/1*0-fsK9RkqL3rogo2SnZPCg.png

Considering the data points on the right, we present mean squared errors and L^2 norms of the weights for three linear regression models:

Je těžké zjistit správnou hodnotu parametru λ.



(a) #params = 3
MSE = 0.006
L2 norm = 0.90
L1 norm = 0.98

(b) #params = 9
MSE = 0.035
L2 norm = 1.06
L1 norm = 2.32

(c) #params = 9
MSE = 0
L2 norm = 32.69
L1 norm = 70.03

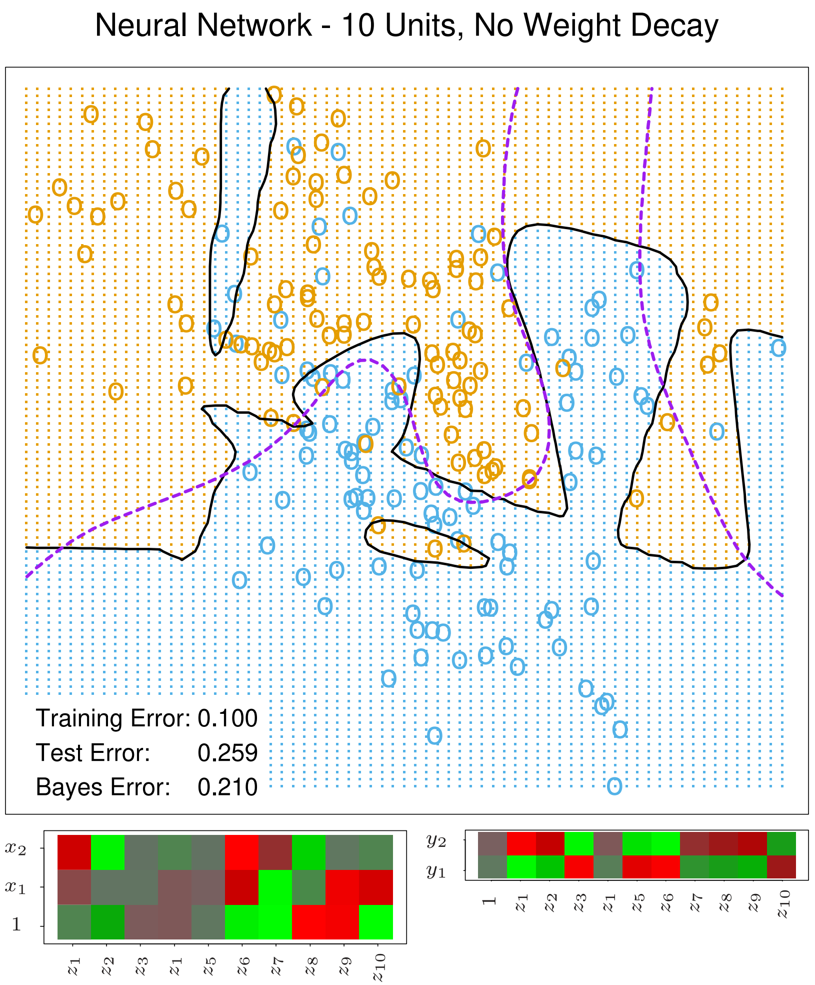
https://miro.medium.com/max/2880/1*DVFYChNDMNIS_7CVq2PhSQ.png

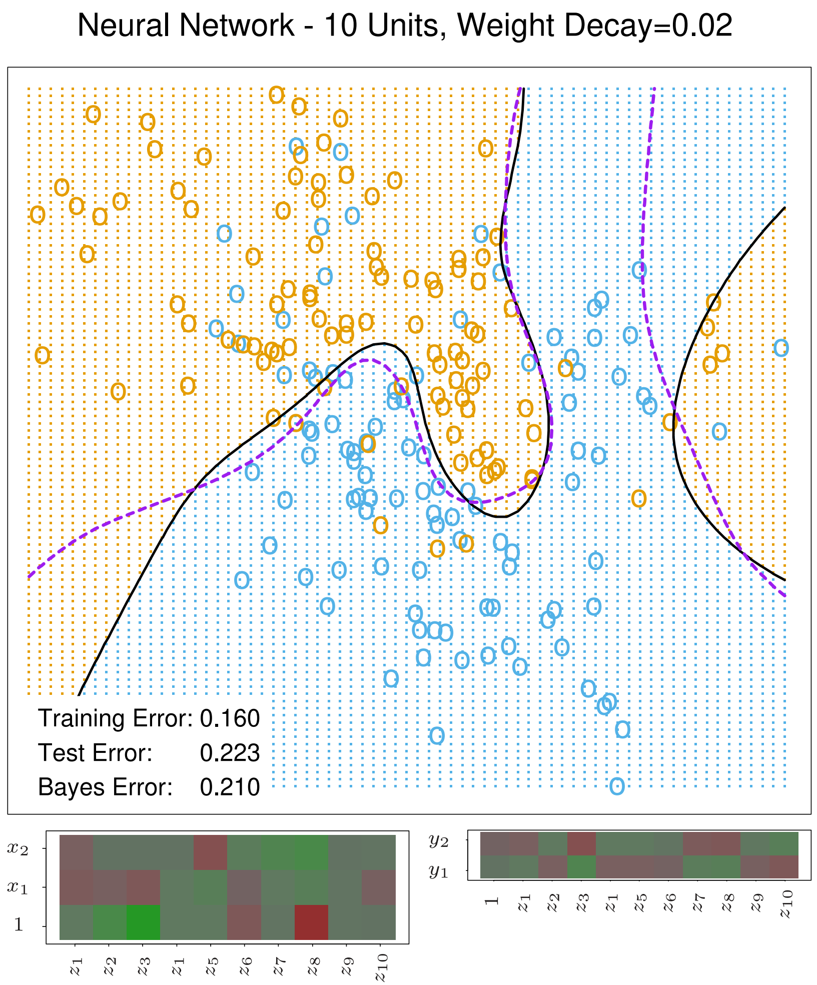
menší MSE neznamená kvalitu!

Figure a: $\hat{y} = 0.04 + 0.04x + 0.9x^2$
Figure b: $\hat{y} = -0.01 + 0.01x + 0.8x^2 + 0.5x^3 - 0.1x^4 - 0.1x^5 + 0.3x^6 - 0.3x^7 + 0.2x^8$
Figure c: $\hat{y} = -0.01 + 0.57x + 2.67x^2 - 4.08x^3 - 12.25x^4 + 7.41x^5 + 24.87x^6 - 3.79x^7 - 14.38x^8$

https://miro.medium.com/max/2880/1*UoIRIKXikCz7SFsPfSZrYQ.png

L2 Regularization





Figures 11.4, 11.5 of "The Elements of Statistical Learning: Data Mining, Inference, and Prediction", https://hastie.su.domains/ElemStatLearn/

L2 Regularization as MAP

Another way to arrive at L^2 regularization is to utilize Bayesian inference.

With MLE we have

$$\boldsymbol\theta_\text{MLE} = \arg\max_{\boldsymbol\theta} p(\mathbb{X}; \boldsymbol\theta).$$

Instead, we may want to maximize **maximum a posteriori (MAP)** point estimate:

$$\boldsymbol\theta_\text{MAP} = \arg\max_{\boldsymbol\theta} p(\boldsymbol\theta|\mathbb{X}).$$

Using Bayes' theorem stating that

$$p(\boldsymbol\theta|\mathbb{X}) = \frac{p(\mathbb{X}|\boldsymbol\theta)p(\boldsymbol\theta)}{p(\mathbb{X})},$$

tohle je fixní vzhledem k θ, takže to vyhodím

we can rewrite the MAP estimate to

tím že chci ty nejmenší váhy.. $$\boldsymbol\theta_\text{MAP} = \arg\max_{\boldsymbol\theta} p(\mathbb{X}|\boldsymbol\theta)p(\boldsymbol\theta).$$ tohle vyjadřuje moji prioritu na θ.

L2 Regularization as MAP

The $p(\boldsymbol\theta)$ are prior probabilities of the parameter values (our *preference*).

A common choice of the preference is the *small weights preference*, where the mean is assumed to be zero, and the variance is assumed to be σ^2. Given that we have no further information, we employ the maximum entropy principle, which results in $p(\theta_i) = \mathcal{N}(\theta_i; 0, \sigma^2)$, so that $p(\boldsymbol\theta) = \prod_i \mathcal{N}(\theta_i; 0, \sigma^2) = \mathcal{N}(\boldsymbol\theta; \mathbf{0}, \sigma^2 \boldsymbol{I})$. Then

$$\begin{aligned}\boldsymbol\theta_\mathrm{MAP} &= \arg\max\nolimits_{\boldsymbol\theta} p(\mathbb{X}; \boldsymbol\theta) p(\boldsymbol\theta) \\ &= \arg\max\nolimits_{\boldsymbol\theta} \prod\nolimits_{i=1}^m p(\boldsymbol x^{(i)}; \boldsymbol\theta) p(\boldsymbol\theta) \\ &= \arg\min\nolimits_{\boldsymbol\theta} \sum\nolimits_{i=1}^m \Big(-\log p(\boldsymbol x^{(i)}; \boldsymbol\theta) - \log p(\boldsymbol\theta)\Big).\end{aligned}$$

By substituting the probability of the Gaussian prior, we get

$$\boldsymbol\theta_\mathrm{MAP} = \arg\min_{\boldsymbol\theta} \sum_{i=1}^m \Big(-\log p(\boldsymbol x^{(i)}; \boldsymbol\theta) + \frac{\#\boldsymbol\theta}{2}\log(2\pi\sigma^2) + \frac{\|\boldsymbol\theta\|_2^2}{2\sigma^2}\Big).$$

L2 Regularization

The resulting parameter update during SGD with L^2-regularization is

$$\theta_i \leftarrow \theta_i - \alpha\frac{\partial E}{\partial \theta_i} - \alpha\lambda\theta_i,\ \ \text{or in vector notation,}\ \ \boldsymbol\theta \leftarrow \boldsymbol\theta - \alpha\nabla_{\boldsymbol\theta} E(\boldsymbol\theta) - \alpha\lambda\boldsymbol\theta.$$

This update can be rewritten to

$$\theta_i \leftarrow \underline{\theta_i(1-\alpha\lambda)} - \alpha\frac{\partial E}{\partial \theta_i},\ \ \text{or in vector notation,}\ \ \boldsymbol\theta \leftarrow \boldsymbol\theta(1-\alpha\lambda) - \alpha\nabla_{\boldsymbol\theta} E(\boldsymbol\theta).$$

$\longleftarrow \to$ v každým kroku ty váhy alespoň maličko zmenšuju.

Termilogically, the update of weights in these two formulas is called *weight decay*, because the weights are multiplied by a factor $1-\alpha\lambda<1$, while adding the L^2-norm of the parameters to the loss is called L^2*-regularization*.

For SGD, they are equivalent – but once you add momentum or normalization by the estimated second moment (RMSProp, Adam), weight decay and L^2-regularization are different.

L2 Regularization – AdamW

It has taken more than three years to realize that using Adam with L^2-regularization does not work well. At the end of 2017, **AdamW** was proposed, which is Adam with weight decay.

Adam with L^2-regularization, AdamW = AdamWeight-decay

```
• $\boldsymbol{s} \leftarrow \mathbf{0}$, $\boldsymbol{r} \leftarrow \mathbf{0}$, $t \leftarrow 0$
• Repeat until stopping criterion is met:
    ○ Sample a minibatch of $m$ training examples $(\boldsymbol{x}^{(i)}, y^{(i)})$
    ○ $\boldsymbol{g} \leftarrow \frac{1}{m}\sum_i \nabla_{\boldsymbol{\theta}}\big(L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|^2\big)$
    ○ $t \leftarrow t + 1$
    ○ $\boldsymbol{s} \leftarrow \beta_1 \boldsymbol{s} + (1-\beta_1)\boldsymbol{g}$
    ○ $\boldsymbol{r} \leftarrow \beta_2 \boldsymbol{r} + (1-\beta_2)\boldsymbol{g}^2$
    ○ $\hat{\boldsymbol{s}} \leftarrow \boldsymbol{s}/(1-\beta_1^t)$, $\hat{\boldsymbol{r}} \leftarrow \boldsymbol{r}/(1-\beta_2^t)$
    ○ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha_t}{\sqrt{\hat{\boldsymbol{r}}}+\varepsilon}\hat{\boldsymbol{s}} \boxed{- \alpha_t\lambda\boldsymbol{\theta}}$
```

takle je blbě, protože to ovlivňuje i momenty, takže se to tam pak bije.

$\alpha \cdot \frac{\hat{s}}{\sqrt{\hat{r}}}$ – poměr mezi první a druhou mocninou

L2 Regularization – AdamW

$$\boldsymbol\theta \leftarrow \boldsymbol\theta - \frac{\alpha_t}{\sqrt{\hat{\boldsymbol r}} + \varepsilon}\hat{\boldsymbol s} - \alpha_t\lambda\boldsymbol\theta$$

In some variants of the algorithm (notably in the original AdamW paper), the authors proposed not to use the learning rate in the weight decay (to decouple the influence of the learning rate on the weight decay).

However, this would mean that if you utilize learning rate decay, you would need to apply it manually also on the weight decay. So currently, the implementation of `keras.optimizers.AdamW` and `torch.optim.AdamW` multiplies the (possibly decaying) learning rate and the (constant) weight decay in the update.

L1 Regularization

Similar to L^2-regularization, but could prefer low L^1 metric of parameters. We could therefore minimize

$$\tilde{E}(\boldsymbol{\theta}; \mathbb{X}) = E(\boldsymbol{\theta}; \mathbb{X}) + \lambda \|\boldsymbol{\theta}\|_1.$$

The corresponding SGD update is then

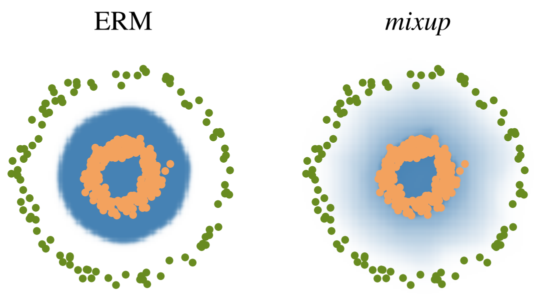
$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial EJ}{\partial \theta_i} - \min\big(\alpha\lambda, |\theta_i|\big)\,\mathrm{sign}(\theta_i).$$

Empirically, L^1-regularization does not work well with deep neural networks and is essentially never used, as far as I know.

Regularization – Dataset Augmentation

For some data, it is cheap to generate slightly modified examples.

- Image processing: translations, horizontal flips, scaling, rotations, color adjustments, …
  - RandAugment
  - Random erasing
  - Mixup (appeared in 2017)



(b) Effect of *mixup* on a toy problem.

Figure 1b of "mixup: Beyond Empirical Risk Minimization", https://arxiv.org/abs/1710.09412

  - CutMix
- Speech recognition: noise, frequency change, …
- More difficult for discrete domains like text.

Regularization – Ensembling

Ensembling (also called **model averaging** or in some contexts *bagging*) is a general technique for reducing generalization error by combining several models. The models are usually combined by averaging their outputs (either distributions or output values in case of a regression).

The main idea behind ensembling it that if models have uncorrelated (independent) errors, then by averaging model outputs the errors will cancel out. If we denote the prediction of the i^{th} model on a training example $(\boldsymbol{x}, y)$ as $y_i(\boldsymbol{x}) = y + \varepsilon_i(\boldsymbol{x})$, so that $\varepsilon_i(\boldsymbol{x})$ is the model error on example $\boldsymbol{x}$, the mean square error of the model is $\mathbb{E}\big[(y_i(\boldsymbol{x}) - y)^2\big] = \mathbb{E}\big[\varepsilon_i^2(\boldsymbol{x})\big]$.

Because for uncorrelated identically distributed random values x_i we have

$$\operatorname{Var}\left(\sum \mathrm{x}_i\right) = \sum \operatorname{Var}(\mathrm{x}_i), \quad \operatorname{Var}(a \cdot \mathrm{x}) = a^2 \operatorname{Var}(\mathrm{x}),$$

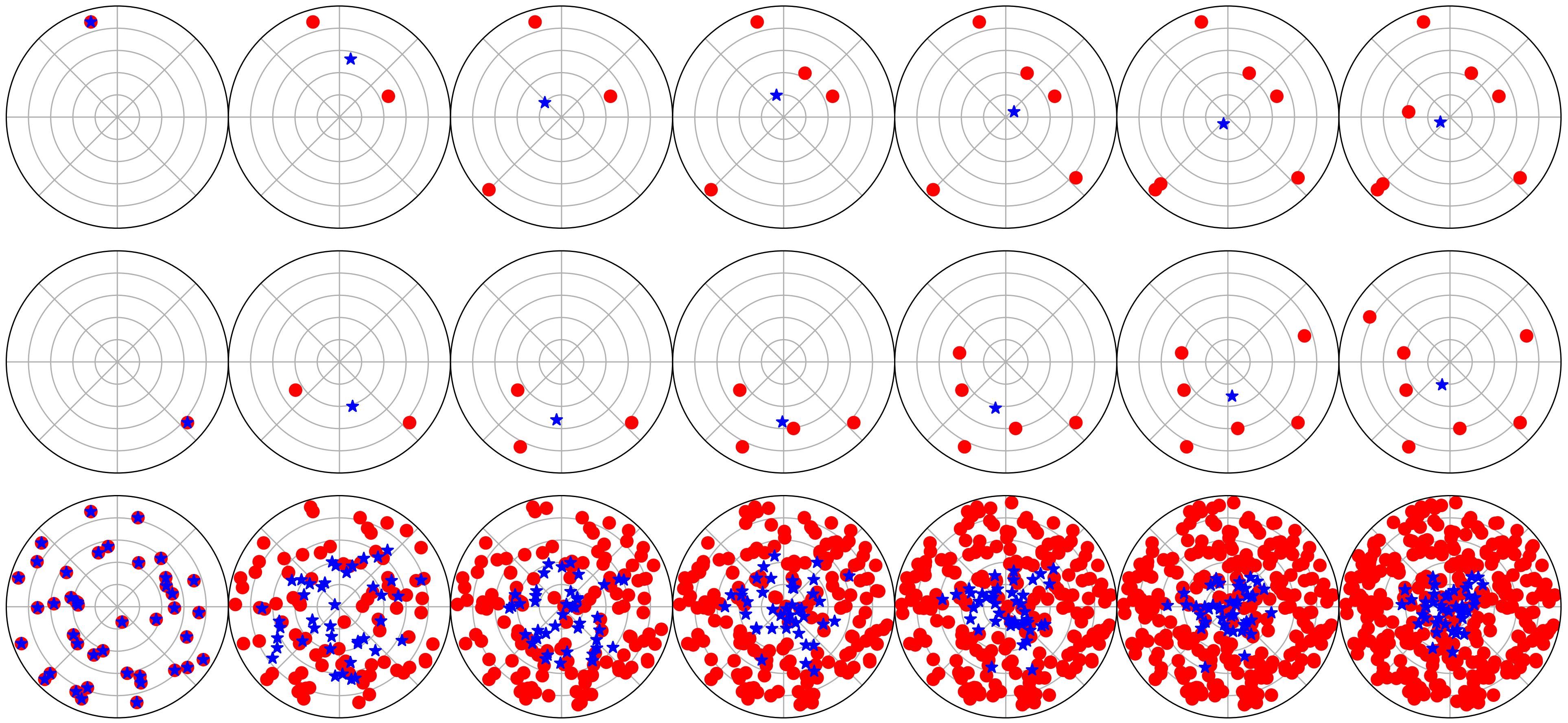
we get that $\operatorname{Var}\left(\frac{1}{n}\sum_i \varepsilon_i\right) = \frac{1}{n}\left(\boxed{\sum_i \frac{1}{n}\operatorname{Var}(\varepsilon_i)}\right)$, so the errors should decrease with the increasing number of models.

průměrná chyba jednotlivých modelů

Čím více modelů mám, tím menší bude chyba.

Regularization – Ensembling Visualization

Consider ensembling predictions generated uniformly on a planar disc:



Regularization – Ensembling

There are many possibilities how to train the models to ensemble:

- For neural network models, training models with independent random initialization is usually enough, given that the loss has many local minima, so the models tend to be quite independent just when using different random initialization.
- Algorithms with convex loss functions usually converge to the same optimum independent of randomization. In that case, we can use **bagging** (bootstrap aggregation), where we generate different training data for each model by sampling with replacement.
- Average models from last hours/days of training.

However, ensembling usually has high performance requirements.

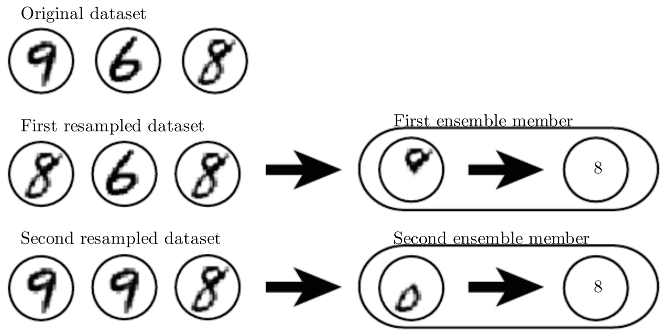
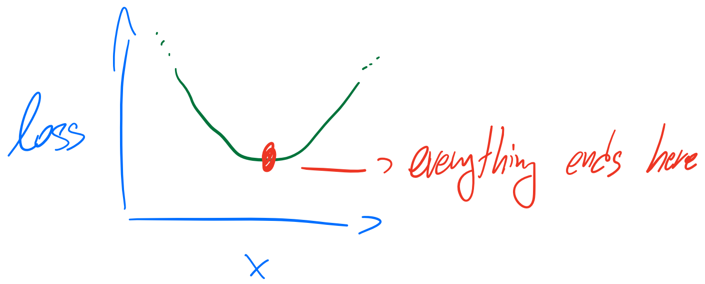


Figure 7.5 of "Deep Learning" book, https://www.deeplearningbook.org



Regularization – Dropout

How to design good universal features?

- In reproduction, evolution is achieved using gene swapping. The genes must not be just good with combination with other genes, they need to be universally good.

Idea of **dropout** by (Srivastava et al., 2014), in preprint since 2012.

When applying dropout to a layer, we drop each neuron independently with a probability of p (usually called **dropout rate**). To the rest of the network, the dropped neurons have value of zero.

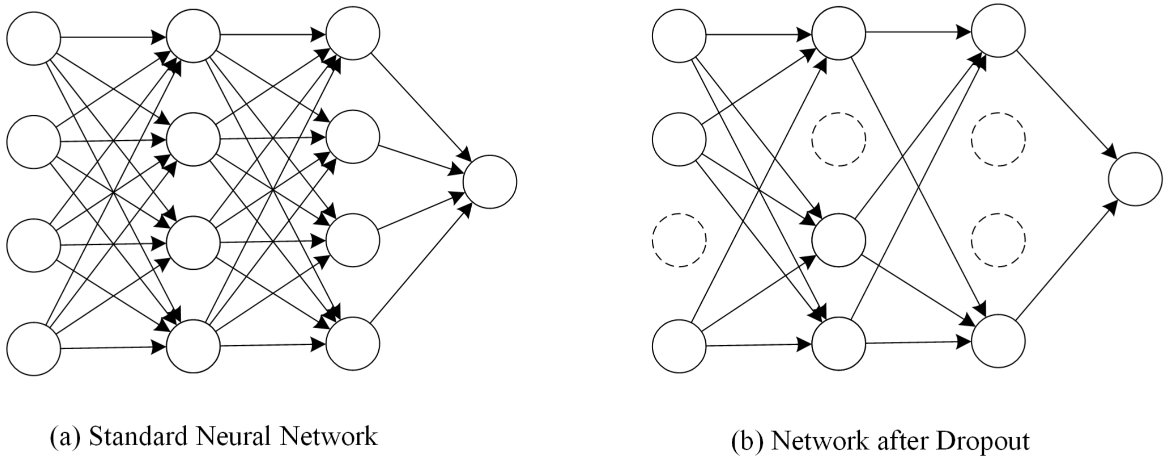
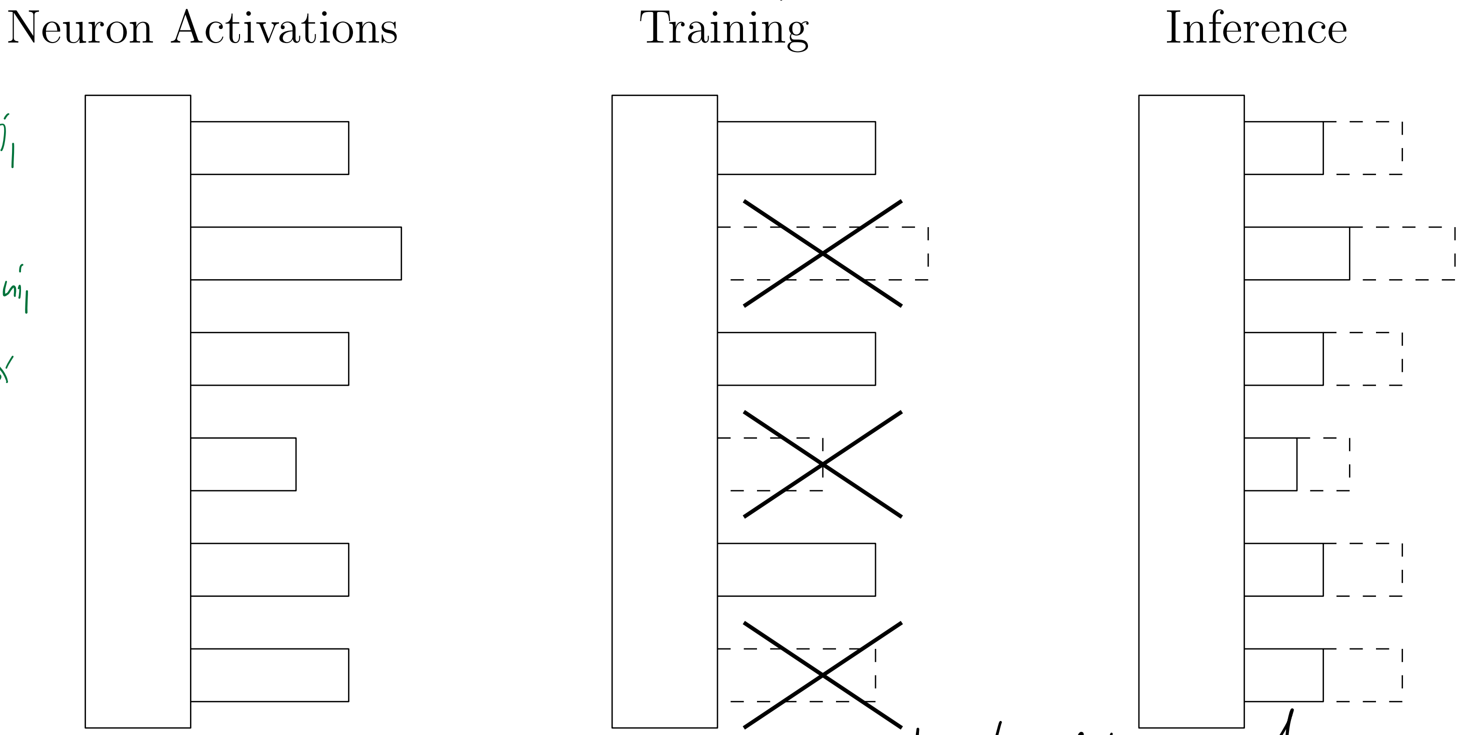


Figure 4 of "Multiple Instance Fuzzy Inference Neural Networks" by Amine B. Khalifa et al.

Regularization – Dropout

Dropout is performed only when training, during inference no nodes are dropped. However, in that case we need to **scale the activations down** by a factor of $1 - p$ to account for more neurons than usual.

musím pak škálovat, jinak dostanu mimo dat



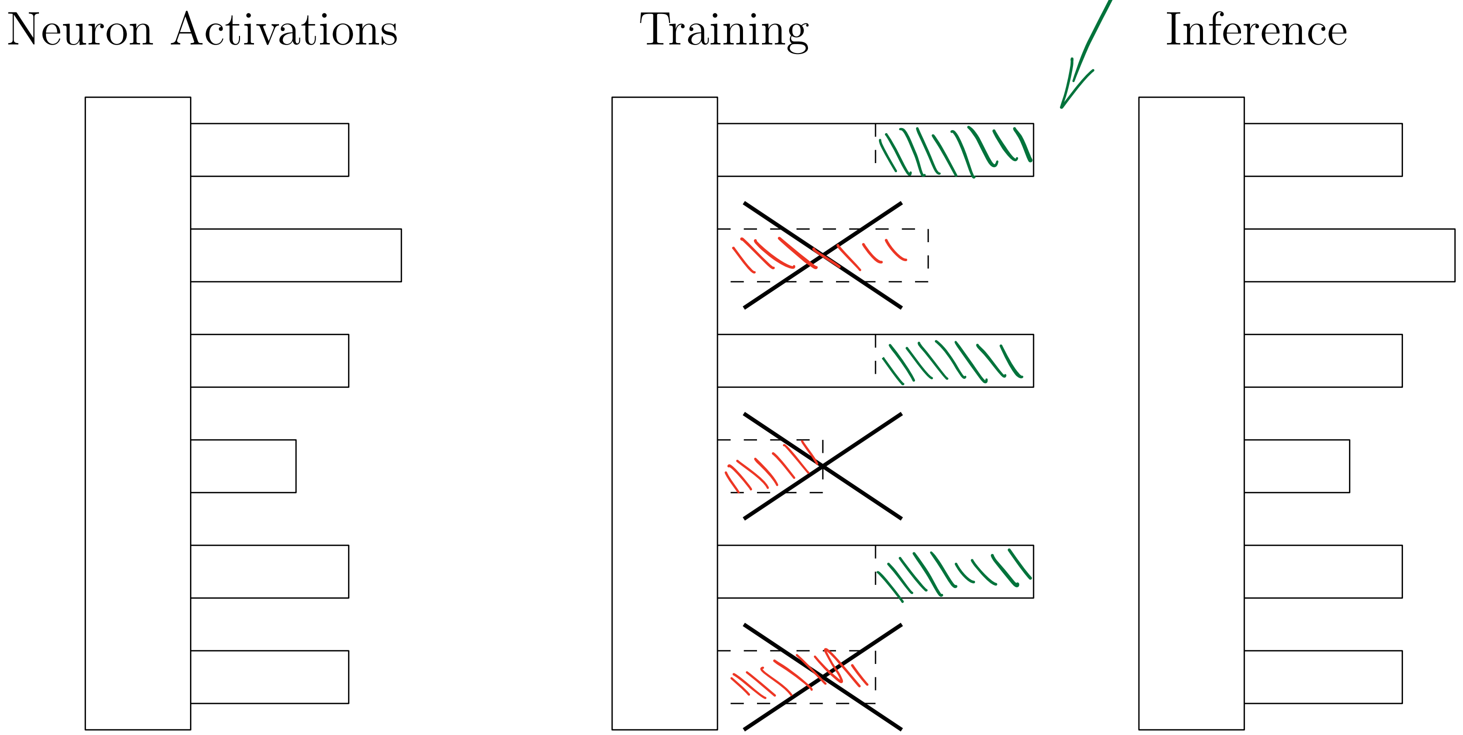
- neurony se naučí, že ne všechny neurony jsou aktivní, takže dají vyšší výstupy a suma by byla příliš vysoká

→ abych to pak už nemusel řešit

→ v realitě ale nafukují o $\frac{1}{1-p}$ ten výstup po drop-outu.

Regularization – Dropout

In practice, the dropout is implemented by instead **scaling the activations up** during training by a factor of $1/(1-p)$ and then **doing nothing** during inference.



Regularization – Dropout as Ensembling

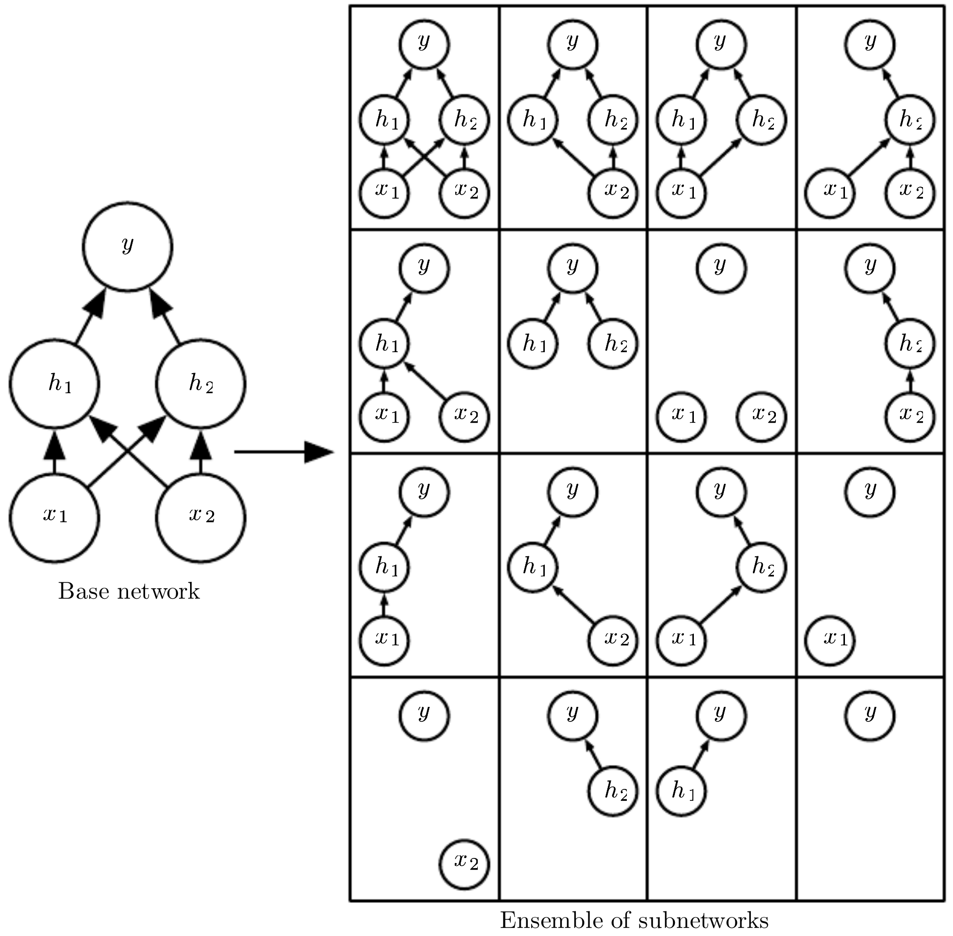


Figure 7.6 of "Deep Learning" book, https://www.deeplearningbook.org

We can understand dropout as a layer obtaining inputs $\boldsymbol{x}$ and multiplying them element-wise by a vector of Bernoulli random variables $\mathbf{z}$, where each z_i is 0 with a probability p:

$$\operatorname{dropout}(\boldsymbol{x}|\mathbf{z}) = \boldsymbol{x} \odot \mathbf{z}.$$

- During training, we sample $\mathbf{z}$ randomly.
- During inference, we compute an expectation over all $\mathbf{z}$:

$$\begin{aligned}\mathbb{E}_{\mathbf{z}}\big[\boldsymbol{x} \odot \mathbf{z}\big] &= p \cdot \boldsymbol{x} \odot \mathbf{0} + (1-p) \cdot \boldsymbol{x} \odot \mathbf{1} \\ &= (1-p) \cdot \boldsymbol{x}.\end{aligned}$$

- In order for the inference to be an identity, we can use $\operatorname{dropout}(\boldsymbol{x}|\mathbf{z}) = \frac{1}{1-p} \cdot \boldsymbol{x} \odot \mathbf{z}$.

Regularization – Dropout Implementation

```
def dropout(inputs, rate=0.5, training=False):
    def do_inference():
        return inputs

    def do_train():
        random_noise = keras.random.uniform(keras.ops.shape(inputs))
        mask = keras.ops.cast(random_noise >= rate, inputs.dtype)
        return inputs * mask / (1 - rate)

    if training:
        return do_train()
    else:
        return do_inference()
```

Vylosené některý hodnoty z výstupu hidden_layer nastavím na 0.

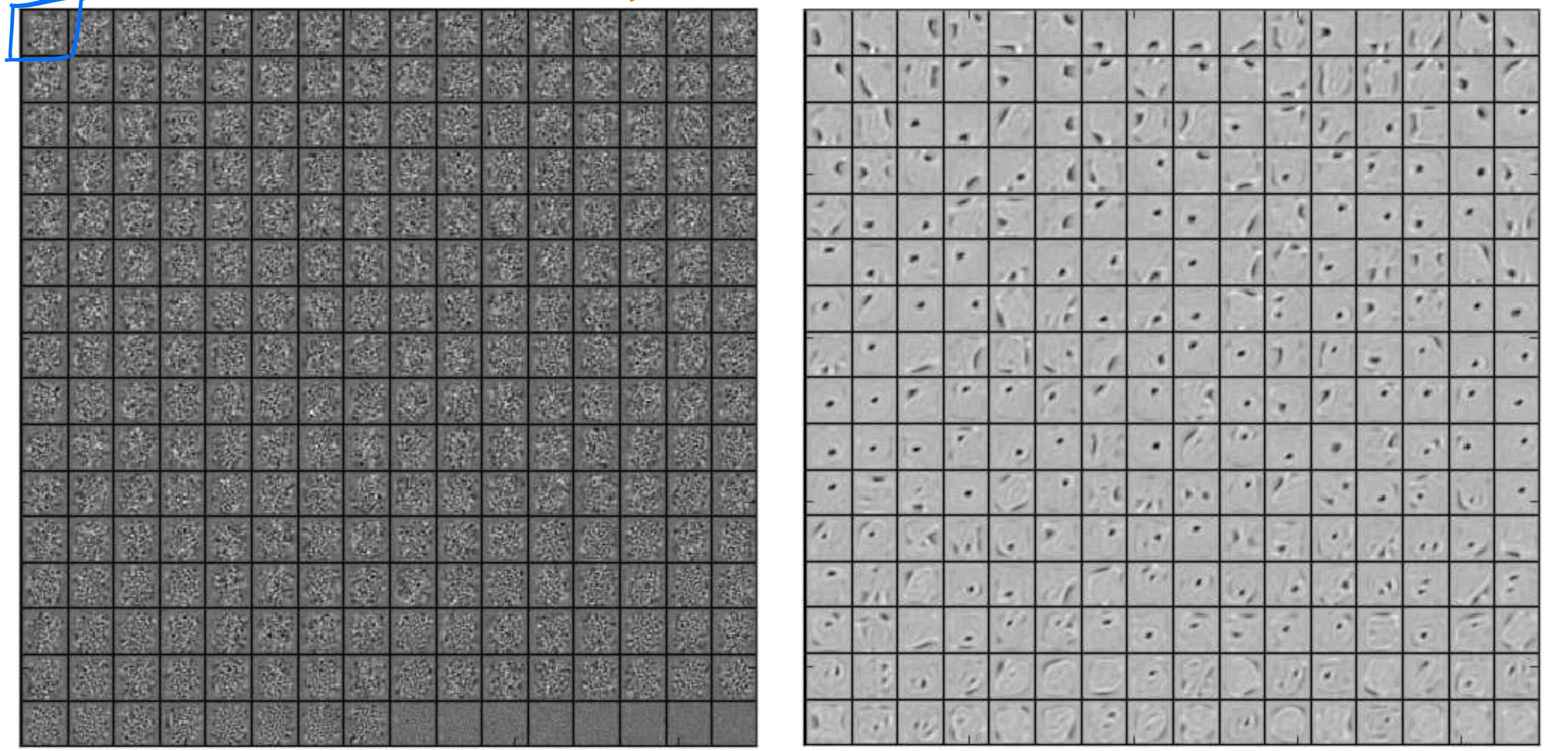
Regularization – Dropout Effect

→ když dobře nakombinuju všechny tyhle váhy, dostanu ty čísla z MNISTu.

matice vah jednoho řádku

tohle jsou váhy pro řádky matice

takhle už každý neuron obsluhuje nějaký kus obrázku. A umím udělat i čísla, co jsem neviděl.



(a) Without dropout (b) Dropout with $p = 0.5$.

tady ale ne tady každý neuron stojí sám za sebe

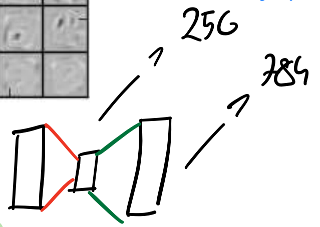


Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.

Figure 7 of "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Regularization – Label Smoothing

Problem with softmax MLE loss is that it is *never satisfied*, always pushing the gold label probability higher (but it saturates near 1).

This behaviour can be responsible for overfitting, because the network is always commanded to respond more strongly to the training examples, not respecting similarity of different training examples.

Ideally, we would like a full (non-sparse) categorical distribution of classes for training examples, but that is usually not available.

We can at least use a simple smoothing technique, called *label smoothing*, which allocates some small probability volume α uniformly for all possible classes.

The target distribution is then

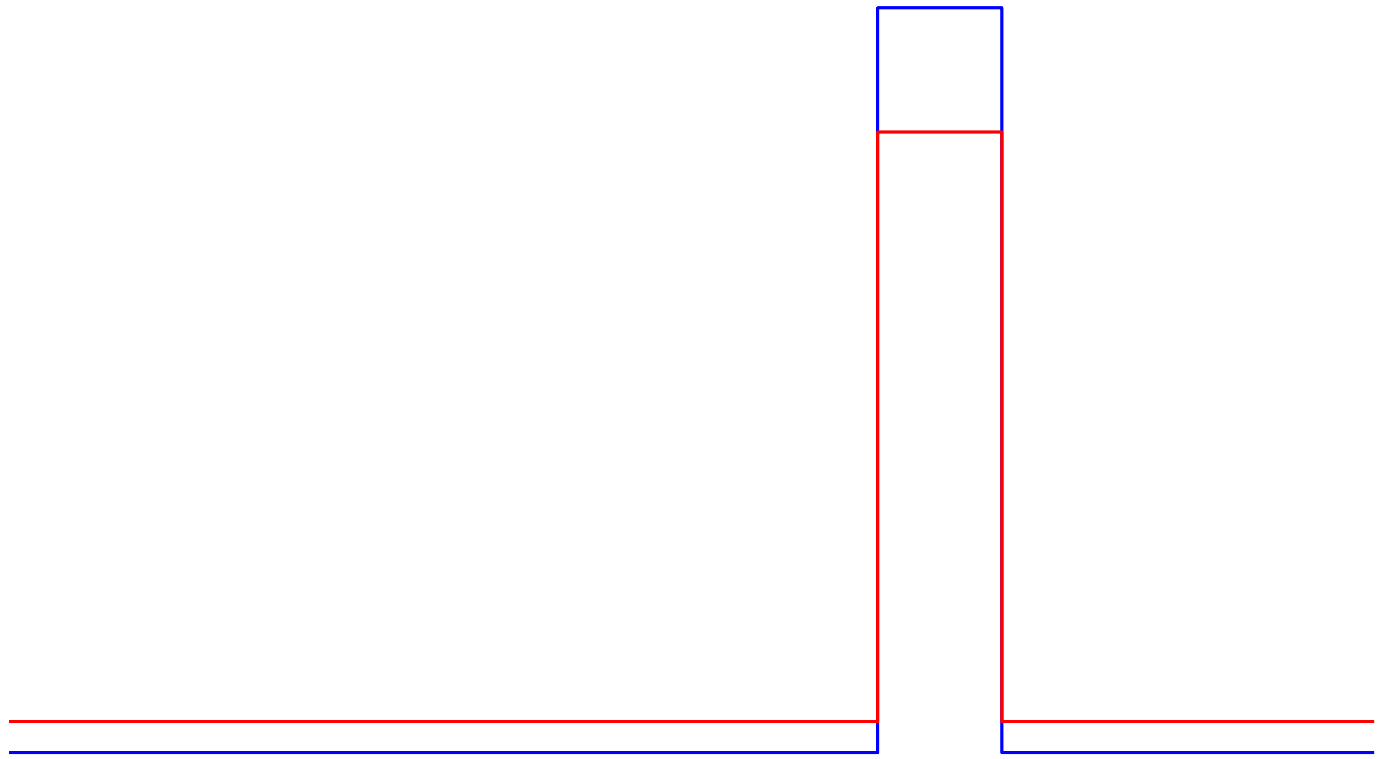
ideální je lepší použít jinou než uniformní distr.

$$(1-\alpha)\mathbf{1}_{gold} + \alpha \frac{\mathbf{1}}{\text{number of classes}}.$$

ale tahle je jednodušší

Správný třídě dám třeba 90% a zbytek dám mezi všechny ostatní třídy. Tím potrestám malé chyby v train datech.

Regularization – Label Smoothing



Gold distribution

Smoothed distribution

Regularization – Good Defaults

When you need to regularize (your model is overfitting), then a good default strategy is to:

- use data augmentation if possible;
- use dropout on all hidden dense layers (not on the output layer), good default dropout rate is 0.5 (or use 0.3-0.1 if the model is underfitting); $\longrightarrow$ čím více, tím méně zaplatím za výkon
- use weight decay (AdamW) for convolutional networks;
- use label smoothing (start with 0.1);
- if you require best performance and have a lot of resources, also perform ensembling.

Convergence

The training process might or might not converge. Even if it does, it might converge slowly or quickly.

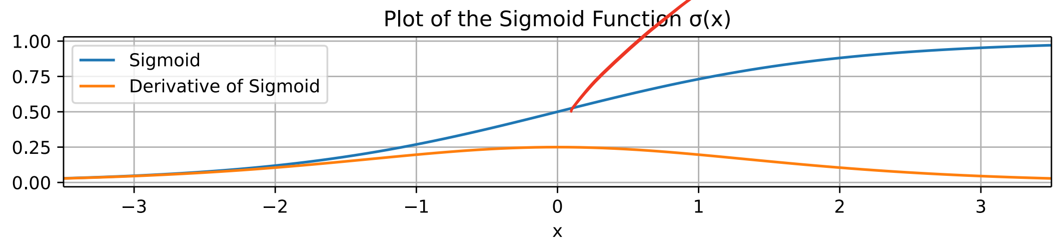
A major issue of convergence of deep networks is to make sure that the gradient with respect to all parameters is reasonable at all times, i.e., it does not decrease or increase too much with depth or in different batches.

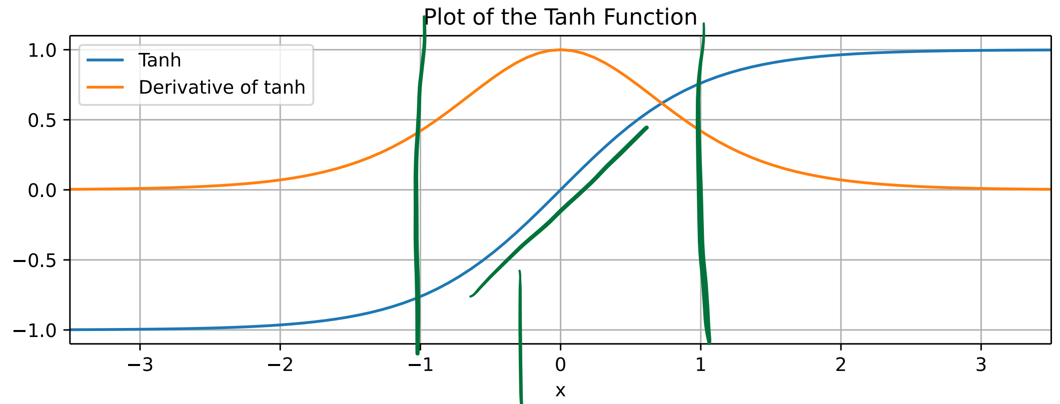
There are *many* factors influencing the gradient, convergence and its speed, we now mention three of them:

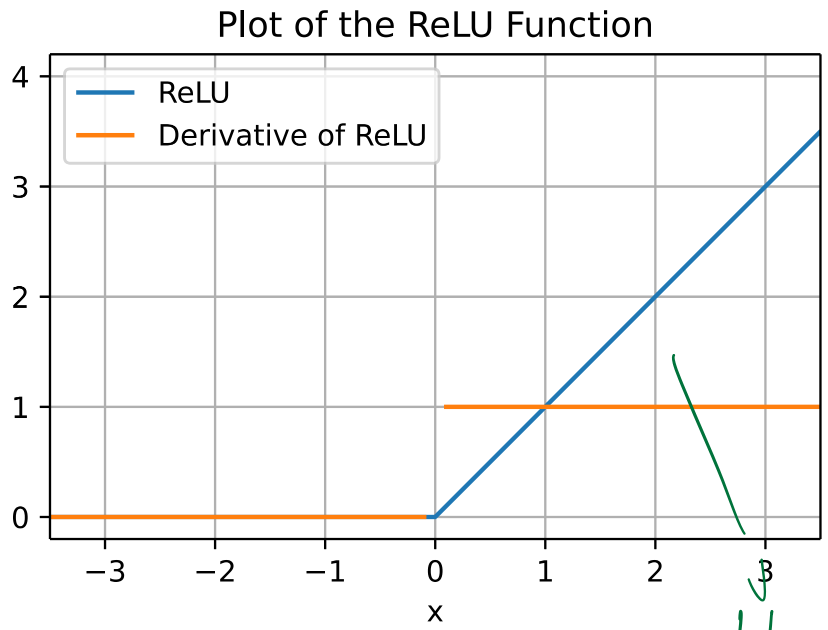
- saturating nonlinearities,
- parameter initialization strategies,
- gradient clipping.

Convergence – Saturating Non-linearities

→ zničíme ; tady ty dobré váhy sešlapuju, přestože to nechci.







v tomhle okolí je to skoro přímka, takže ta derivace je téměř nula.

tady je to hnusný.

Takže alespoň nad dobrými váhami je násobím 1, takže je dobrá.

Convergence – Parameter Initialization

Neural networks usually need random initialization to *break symmetry*.

- Biases are usually initialized to 0 (Keras, TF, Jax; not PyTorch).
- Weights are usually initialized to small random values, either with uniform or normal distribution.
  - The scale matters for deep networks! → chci mít řádově stejně velké vstupy na vrstvách.
  - Originally, people used $U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$ distribution. → kdyby to bylo normální, tak je malá šance, že dostanu jednu gigantickou hodnotu.
    - Still the default for `torch.nn.Linear`.
  - Xavier Glorot and Yoshua Bengio, 2010: *Understanding the difficulty of training deep feedforward neural networks*.

    The authors theoretically and experimentally show that a suitable way to initialize a $\mathbb{R}^{n\times m}$ matrix is

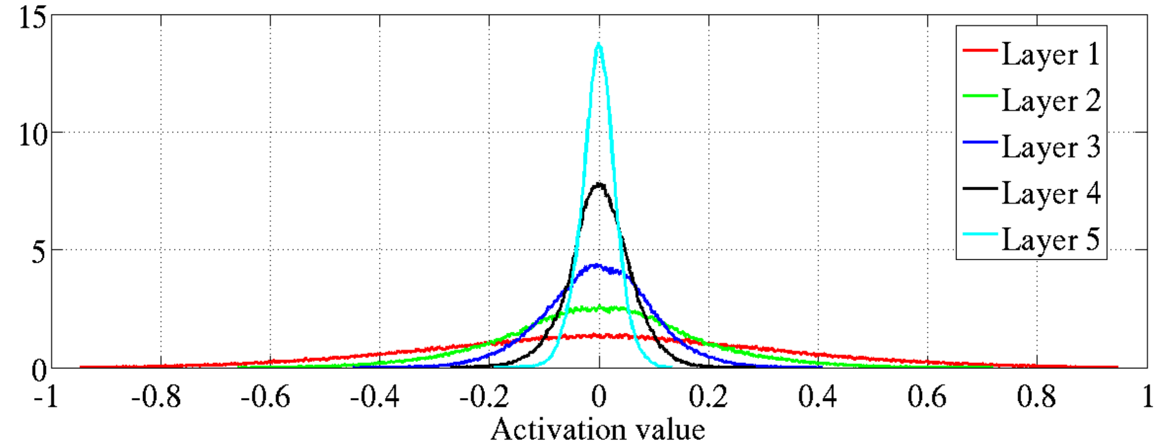
$$U\left[-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right].$$

$= \underline{\sqrt{3}} \cdot \underline{\sqrt{\frac{2}{m+n}}}$ inverz průměrem

→ $x \sim U[-a,a] \Rightarrow Var(x) = \frac{1}{3a}$

tohle se používá všude kromě pytorche default. →

Convergence – Parameter Initialization



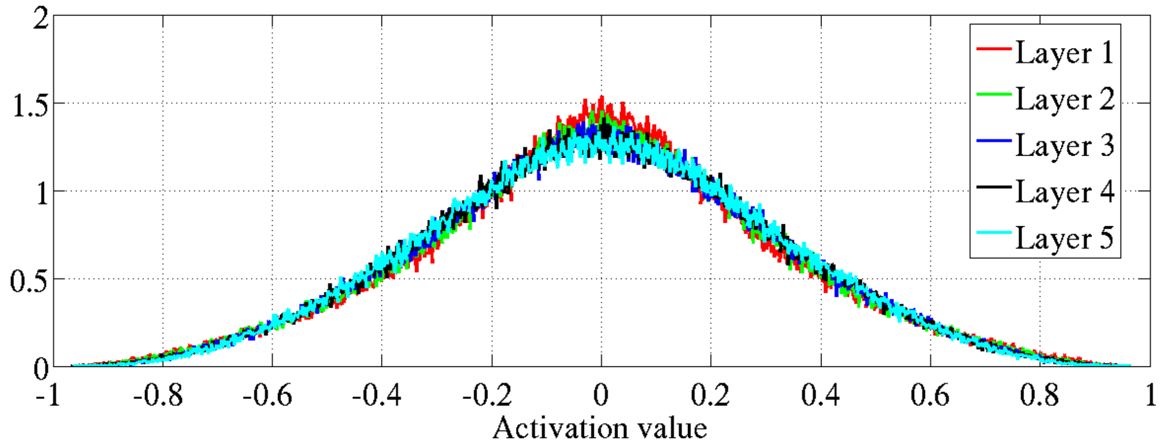


Figure 6 of "Understanding the difficulty of training deep feedforward neural networks", http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Convergence – Parameter Initialization

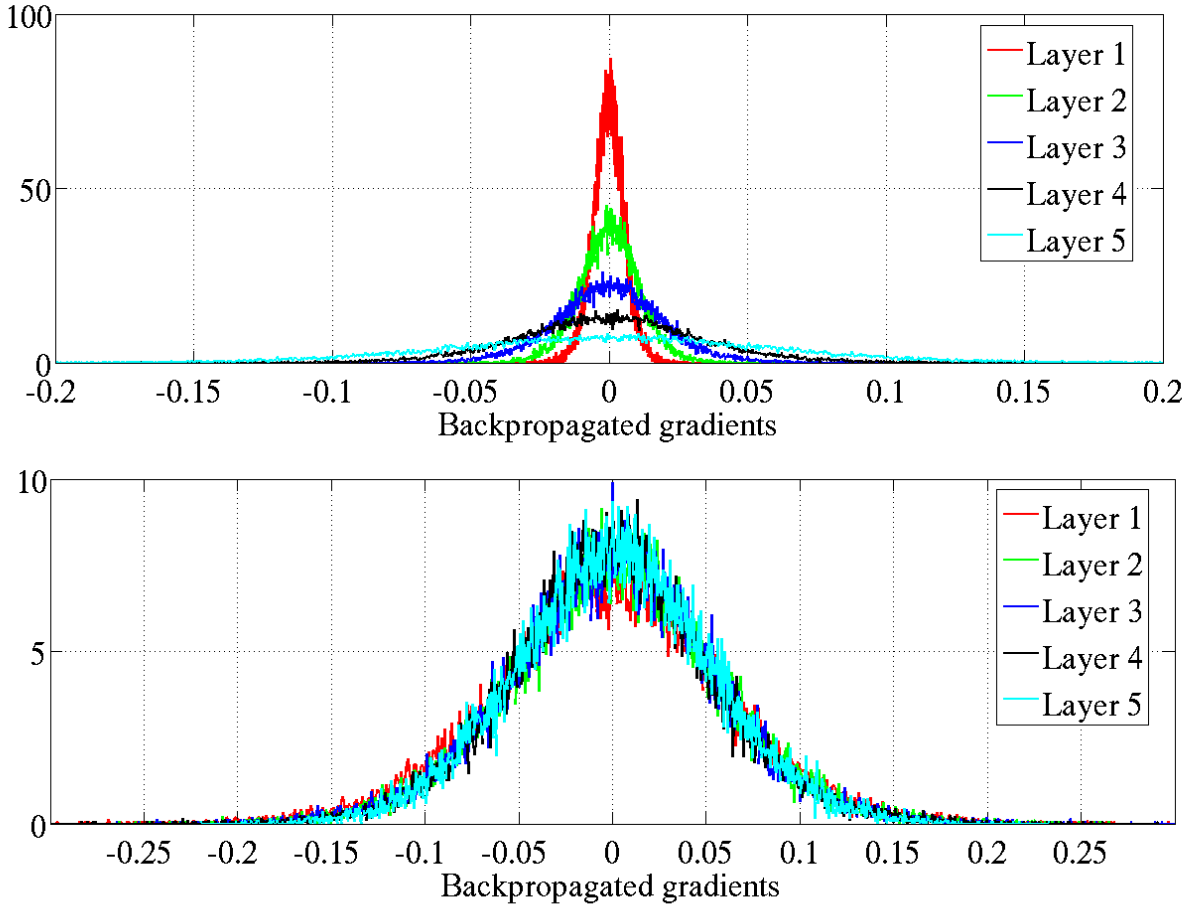


Figure 7 of "Understanding the difficulty of training deep feedforward neural networks", http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Convergence – Gradient Clipping

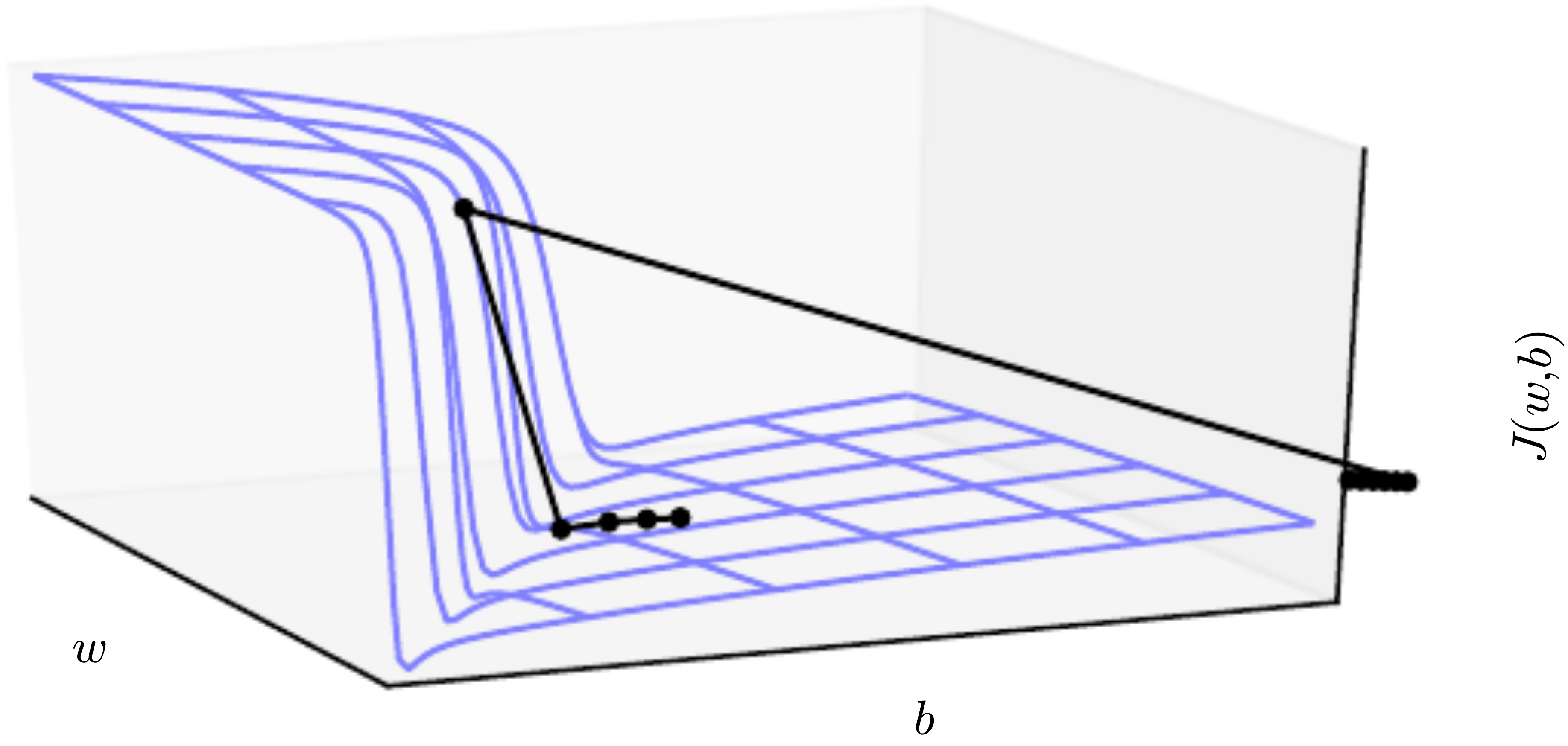


Figure 8.3 of "Deep Learning" book, https://www.deeplearningbook.org

Convergence – Gradient Clipping

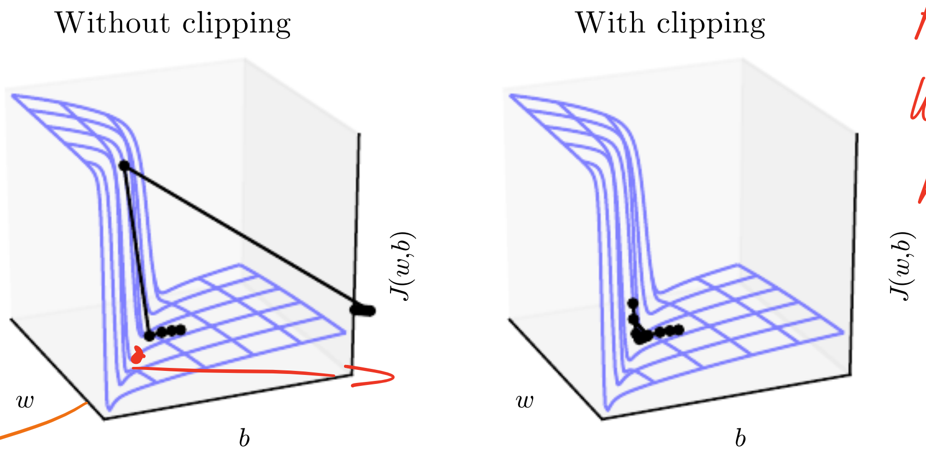


Figure 10.17 of "Deep Learning" book, https://www.deeplearningbook.org

Směr gradientu, jakmile uvidí před sebou tu stěnu.

Adam nepomůže, protože když je ten diff zničeho nic, tak ty momenty budou ještě furt malý a tedy Adam nepomůže.

Using a given maximum norm, we may *clip* the gradient.

$$\boldsymbol{g} \leftarrow \begin{cases} \boldsymbol{g} & \text{if } \|\boldsymbol{g}\| \leq c, \\ c\frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} & \text{if } \|\boldsymbol{g}\| > \underline{c}. \end{cases}$$

1 je dobrý základ

Tedy omezím, jak moc velký skok maximálně může být.

Tzn. že příliš velkým gradientům nevěřím.

Clipping can be performed per weight (param `clipvalue` of `keras.optimizers.Optimizer`), **per variable** (`clipnorm`) or for the gradient as a whole (`global_clipnorm`).

Omezuju tím ale reaktivnost toho modelu na opravdový chyby.

- tedy musím vhodně volit parametr.