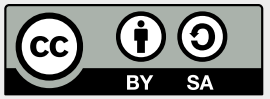
Training Neural Networks II

Milan Straka

March 9, 2020

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



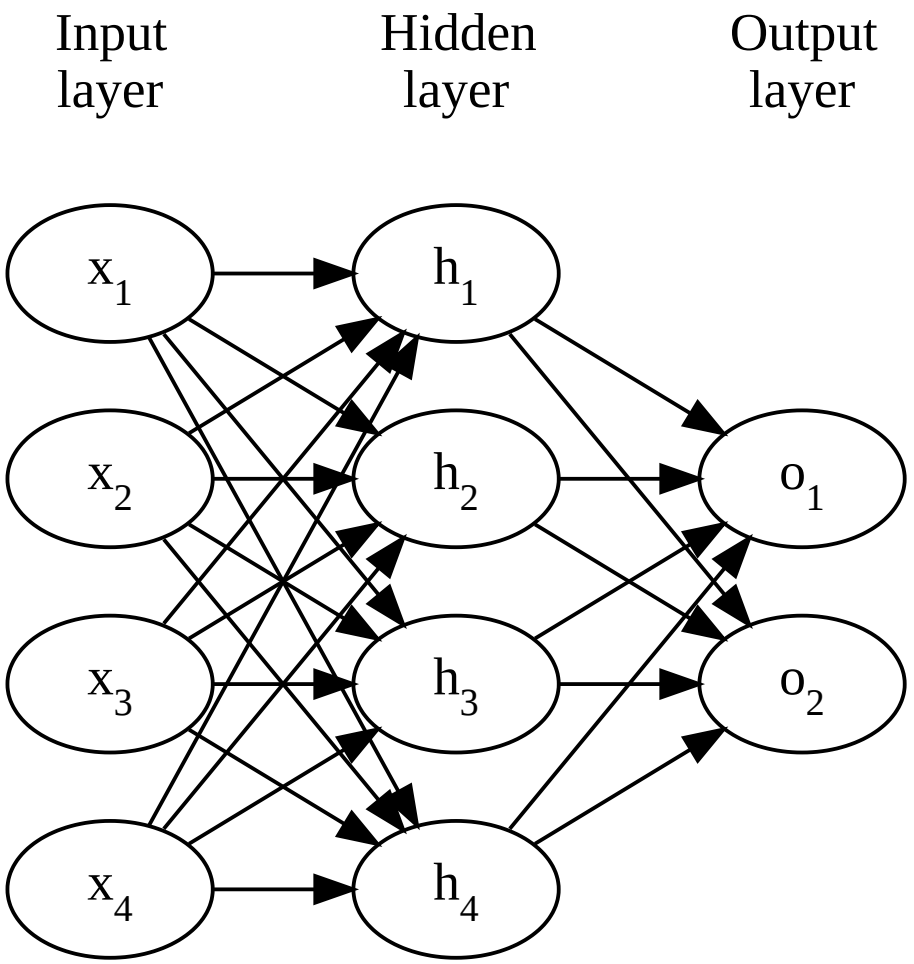
Putting It All Together

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

$$\begin{cases} \text{none} & \text{if } y \in \mathbb{R} \\ \sigma & \text{if } y \text{ is a probability of an outcome} \\ \text{softmax} & \text{if } y \text{ is a gold class} \end{cases}$$

If $\boldsymbol{x} \in \mathbb{R}^d$, we can use a neural network with an input layer of size d, hidden layer of size h with a non-linear activation function, and an output layer of size o (either 1 or number of classification classes) with the mentioned output function.

Putting It All Together



We have

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

where

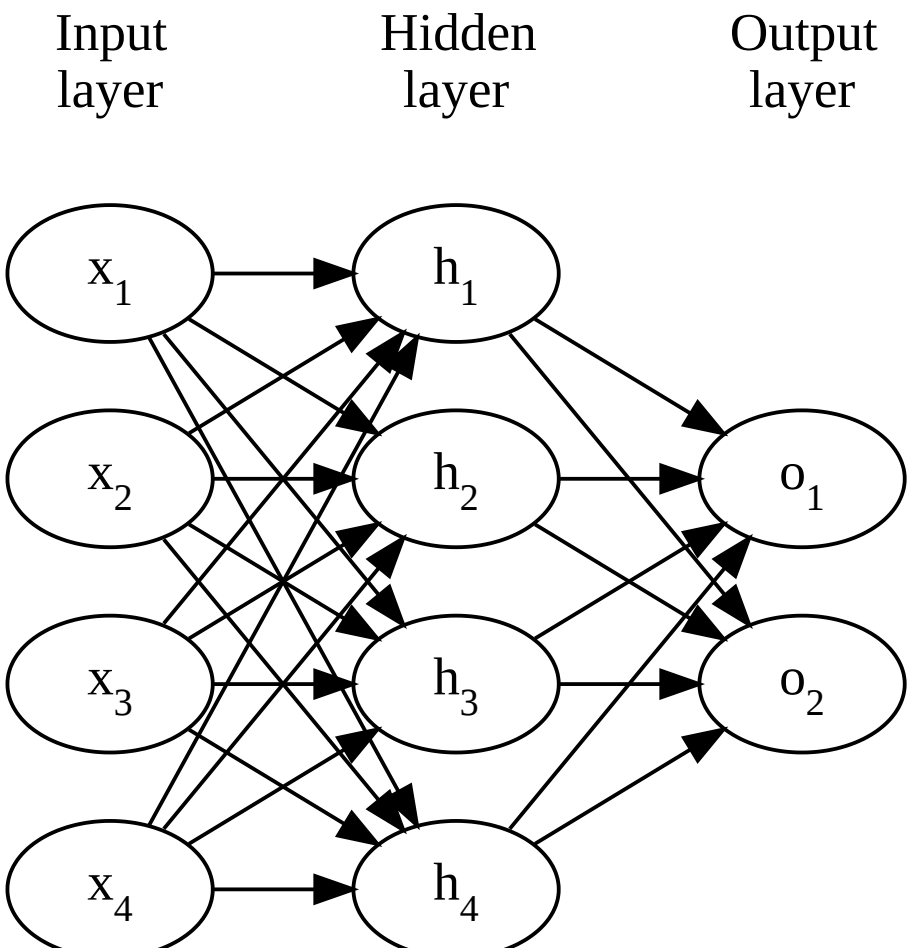
- $\boldsymbol{W}^{(1)} \in \mathbb{R}^{h \times d}$ is a matrix of *weights*,
- $\boldsymbol{b}^{(1)} \in \mathbb{R}^h$ is a vector of *biases*,
- $f^{(1)}$ is an activation function.

The weights are sometimes also called a *kernel*.

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

Transformations of type $wx + b$ are called *affine* instead of *linear*.

Putting It All Together



Similarly

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

with

- $\boldsymbol{W}^{(2)} \in \mathbb{R}^{o \times h}$ 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

The parameters of the model are therefore $\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}$ of total size $d \times h + h \times o + h + o$.

To train the network, we repeatedly sample m training examples and perform SGD (or any its adaptive variant), updating the parameters to minimize the loss.

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial L}{\partial \theta_i}$$

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.

Practical Issues

- Processing all input in *batches*.
- Vector representation of the network.

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

$$\boldsymbol{h} = f^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

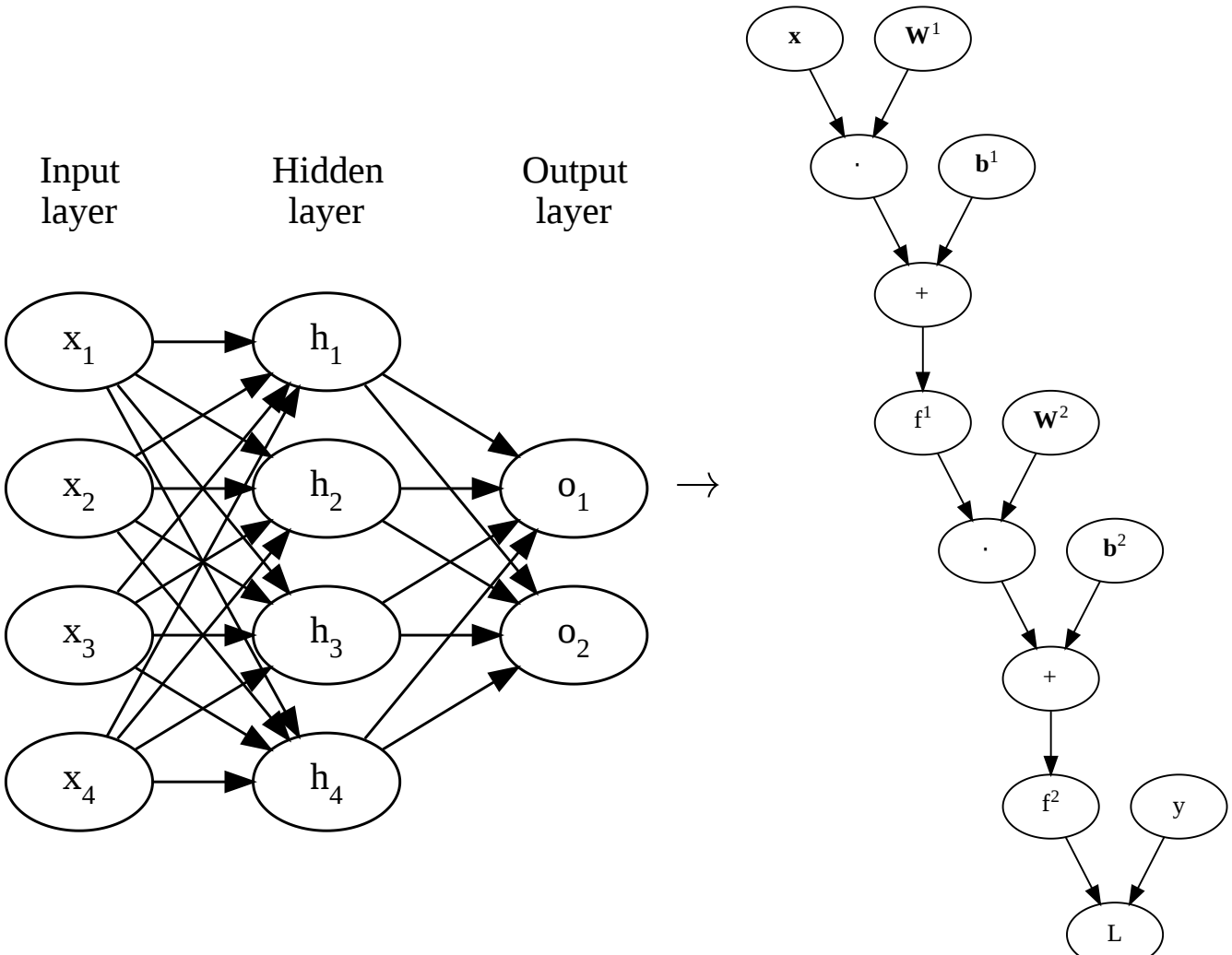
$$\boldsymbol{o} = f^{(2)}\left(\boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}\right) = f^{(2)}\left(\boldsymbol{W}^{(2)}\left(f^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)\right) + \boldsymbol{b}^{(2)}\right)$$

  The derivatives

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

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

Computation Graph



High Level Overview

	Classical ('90s)	Deep Learning
Architecture	⋮⋮⋮	⋮⋮⋮⋮⋮⋮⋮⋮⋮ CNN, RNN, Transformer, VAE, GAN, ...
Activation func.	$\tanh, \sigma$	$\tanh$, ReLU, PReLU, ELU, GELU, Swish, Mish, ...
Output function	none, σ	none, σ, $\mathrm{softmax}$
Loss function	MSE	NLL (or cross-entropy or KL-divergence)
Optimization	SGD, momentum	SGD, RMSProp, Adam, ...
Regularization	L2, L1	L2, Dropout, Label smoothing, BatchNorm, LayerNorm, ...

Derivative of MSE Loss

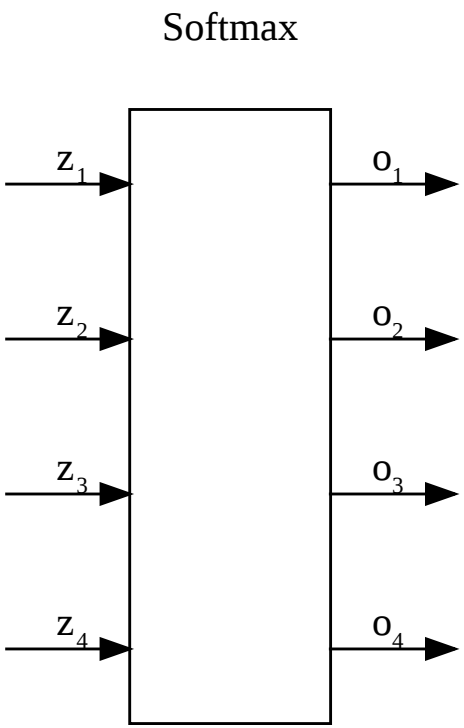
Given the MSE loss of

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

the derivative with respect to $\hat{y}$ is simply:

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

Derivative of Softmax MLE Loss



Let us have a softmax output layer with

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

Derivative of Softmax MLE Loss

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

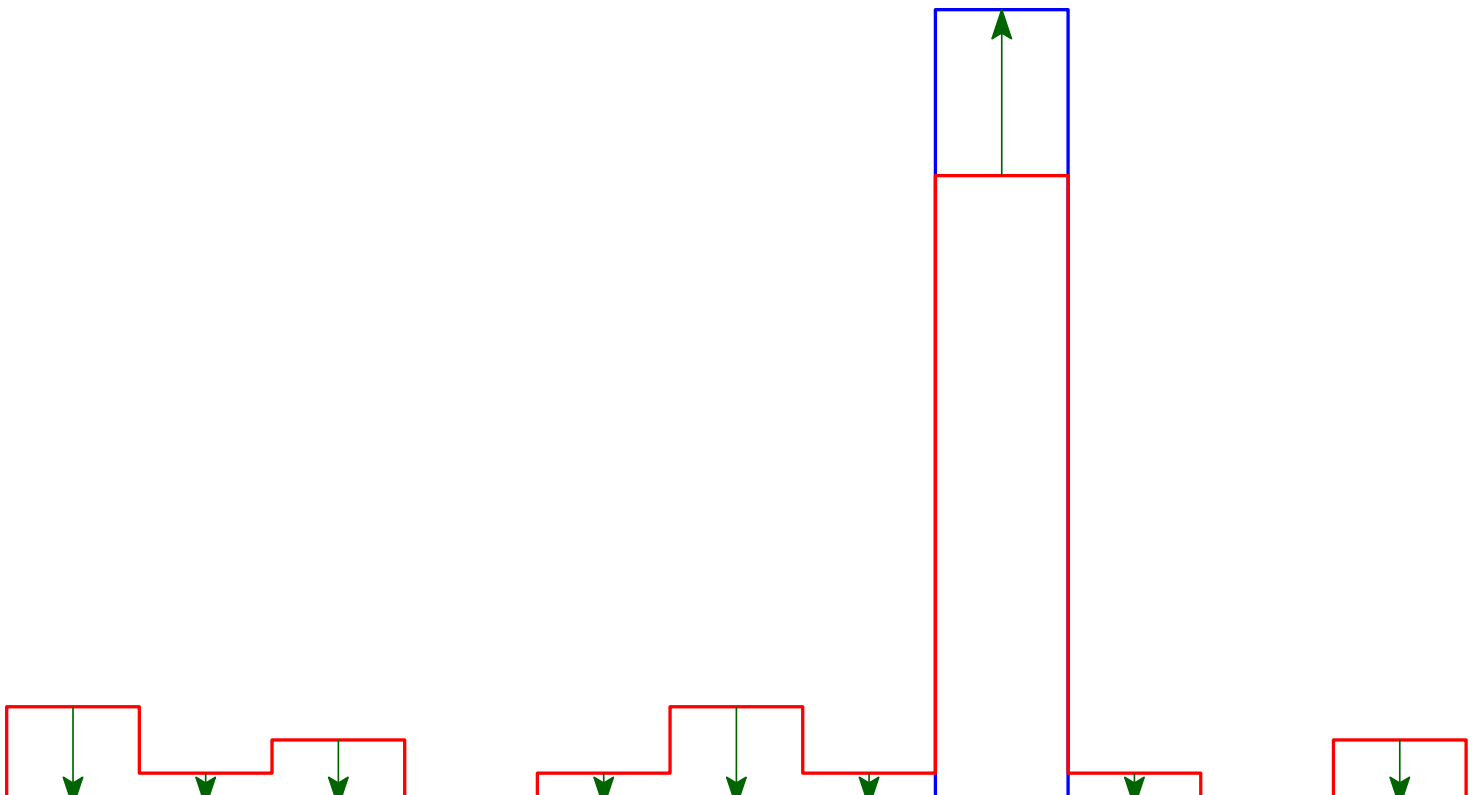
$$L(\mathrm{softmax}(\boldsymbol{z}), gold) = -\log o_{gold}.$$

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

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

Therefore, $\frac{\partial L}{\partial \boldsymbol{z}} = \boldsymbol{o} - \mathbf{1}_{gold}$, where $\mathbf{1}_{gold}$ is 1 at index $gold$ and 0 otherwise.

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

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

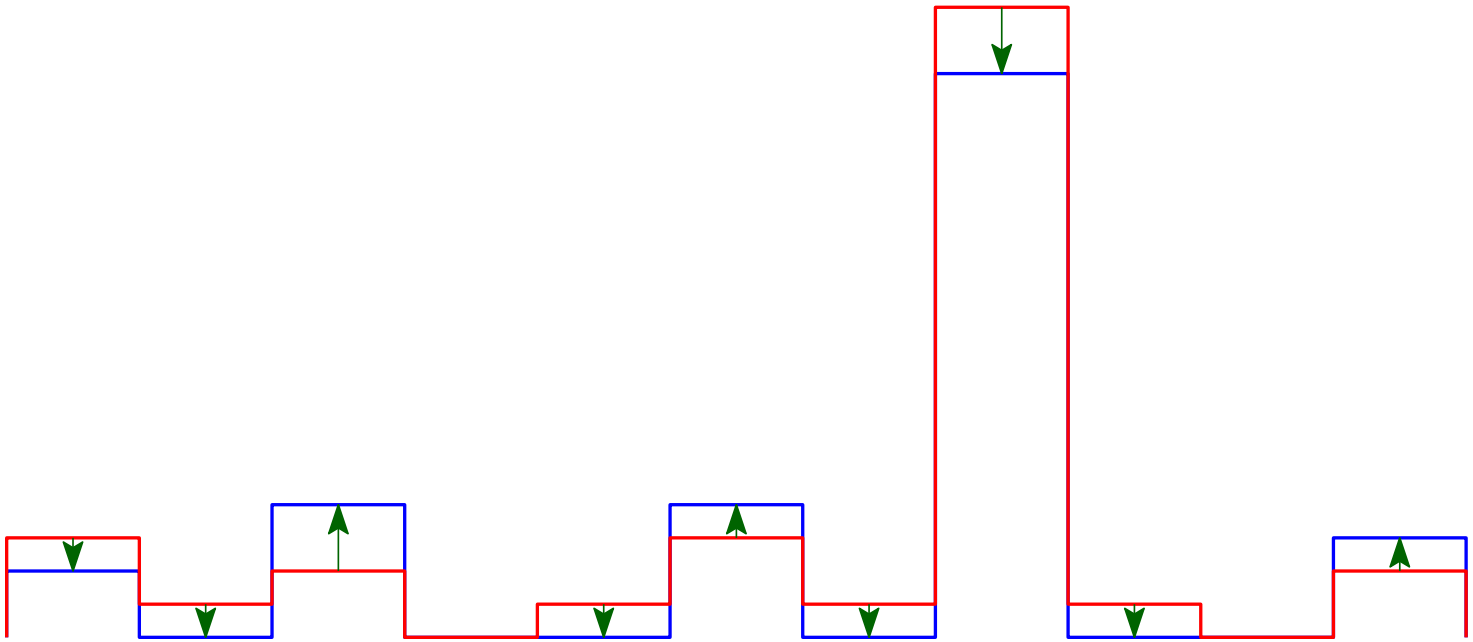
Repeating the previous procedure for each target probability, we obtain

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

Sigmoid

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

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.

- Early stopping
- L2, L1 regularization
- Dataset augmentation
- Ensembling
- Dropout
- Label smoothing

Regularization – Early Stopping

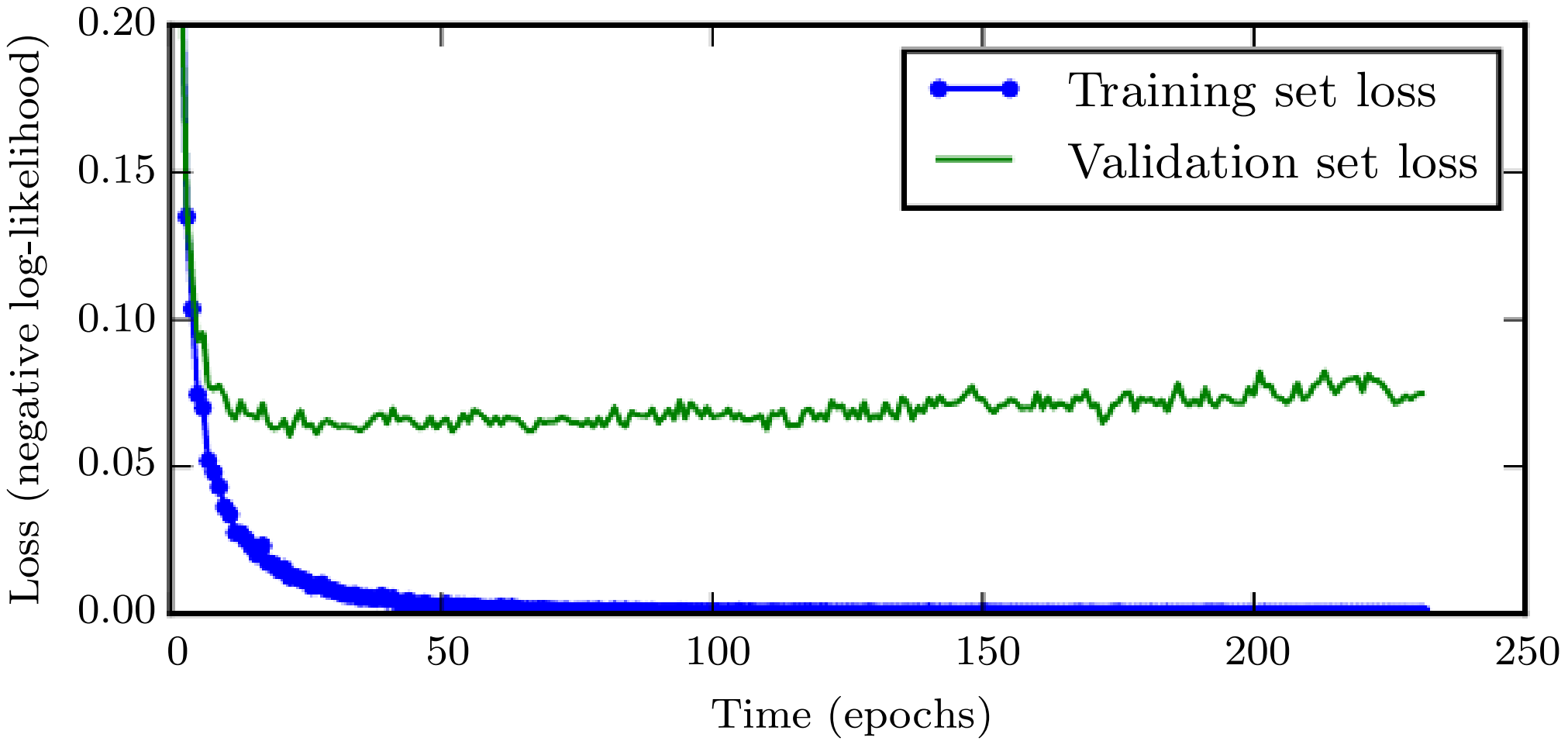


Figure 7.3, page 246 of Deep Learning Book, http://deeplearningbook.org

L2 Regularization

We prefer models with parameters small under L2 metric.

The L2 regularization, also called *weight decay*, *Tikhonov regularization* or *ridge regression* therefore minimizes

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

for a suitable (usually very small) λ.

During the parameter update of SGD, we get

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial J}{\partial \theta_i} - 2\alpha\lambda\theta_i$$

L2 Regularization

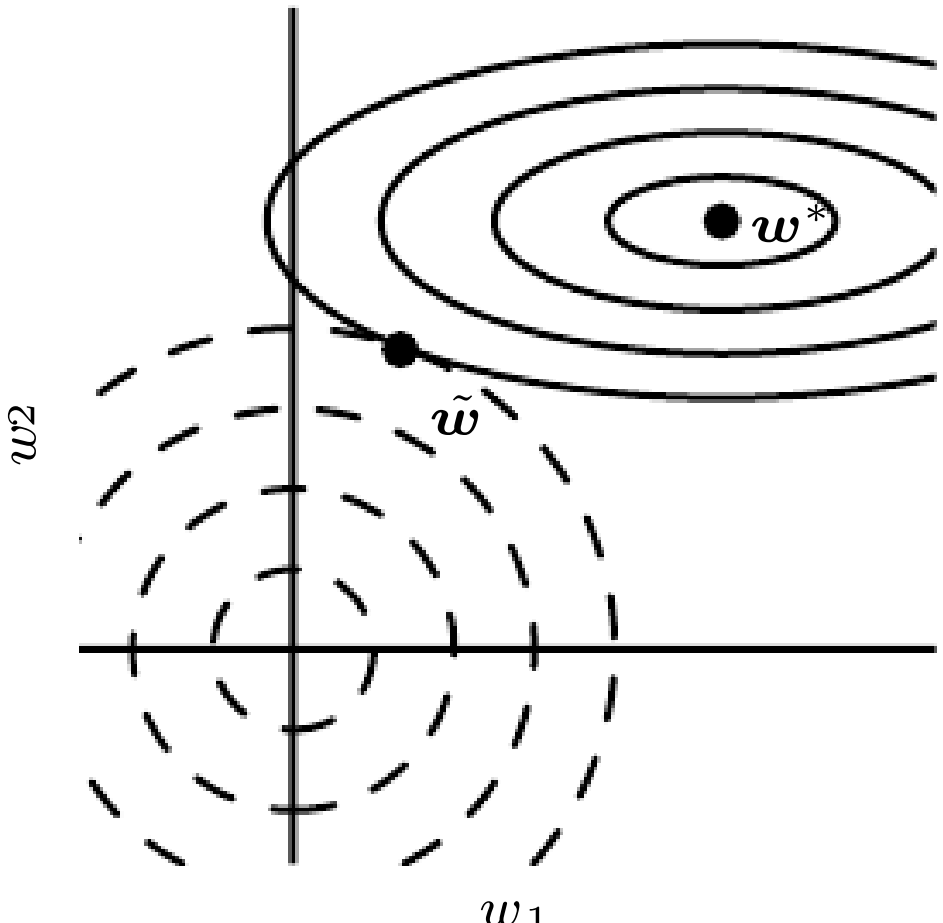


Figure 7.1, page 233 of Deep Learning Book, http://deeplearningbook.org

L2 Regularization as MAP

Another way to arrive at L2 regularization is to utilize Bayesian inference.

With MLE we have

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

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

$$\boldsymbol{\theta}_{\mathrm{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}; \mathbb{X})$$

Using Bayes' theorem

$$p(\boldsymbol{\theta}; \mathbb{X}) = p(\mathbb{X}; \boldsymbol{\theta}) p(\boldsymbol{\theta}) / p(\mathbb{X}),$$

we get

$$\boldsymbol{\theta}_{\mathrm{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\mathbb{X}; \boldsymbol{\theta}) p(\boldsymbol{\theta}).$$

L2 Regularization as MAP

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

One possibility for such a prior is $\mathbb{N}(\boldsymbol{\theta}; 0, \sigma^2)$.

Then

$$\begin{aligned}\boldsymbol{\theta}_{\text{MAP}} &= \arg\max_{\boldsymbol{\theta}} p(\mathbb{X}; \boldsymbol{\theta}) p(\boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) p(\boldsymbol{\theta}) \\ &= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{m} -\log p(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) - \log p(\boldsymbol{\theta})\end{aligned}$$

By substituting the probability of the Gaussian prior, we get

$$\boldsymbol{\theta}_{\text{MAP}} = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{m} -\log p(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) + \frac{1}{2}\log(2\pi\sigma^2) + \frac{\boldsymbol{\theta}^2}{2\sigma^2}$$

L1 Regularization

Similar to L2 regularization, but we prefer low L1 metric of parameters. We therefore minimize

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

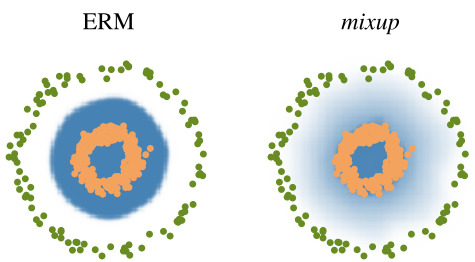
The corresponding SGD update is then

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial J}{\partial \theta_i} - \operatorname{sign}(\theta_i) \alpha \lambda.$$

Regularization – Dataset Augmentation

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

- Image processing: translations, horizontal flips, scaling, rotations, color adjustments, ...
  - Mixup (appeared in 2017)



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

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

- 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.

Because for independent identically distributed random values $\mathbf{x}_i$ we have

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

we get that

$$\mathrm{Var}\left(\frac{1}{n}\sum \mathbf{x}_i\right) = \frac{1}{n}\,\mathrm{Var}(\mathbf{x}_1).$$

However, ensembling usually has high performance requirements.

Regularization – Ensembling

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

- Generate different datasets by sampling with replacement (bagging).

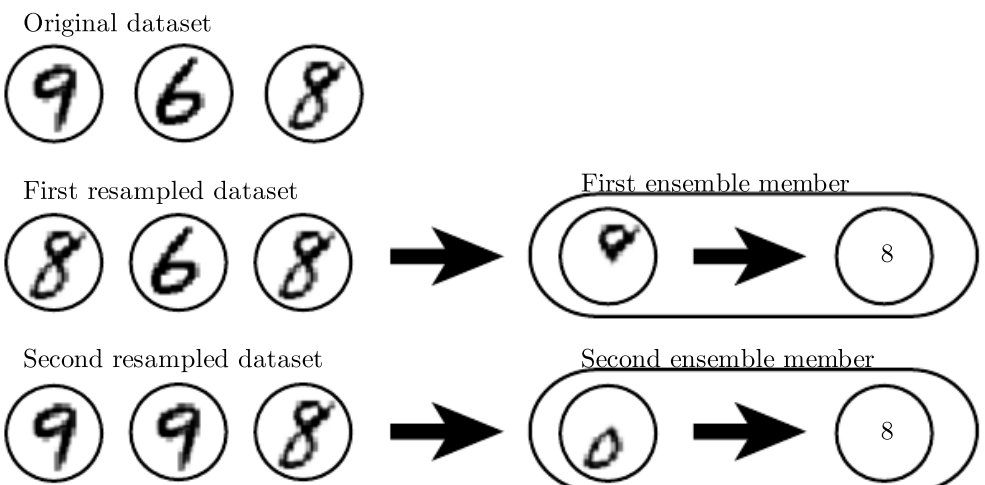


Figure 7.5, page 257 of Deep Learning Book, http://deeplearningbook.org

- Use random different initialization.
- Average models from last hours/days of training.

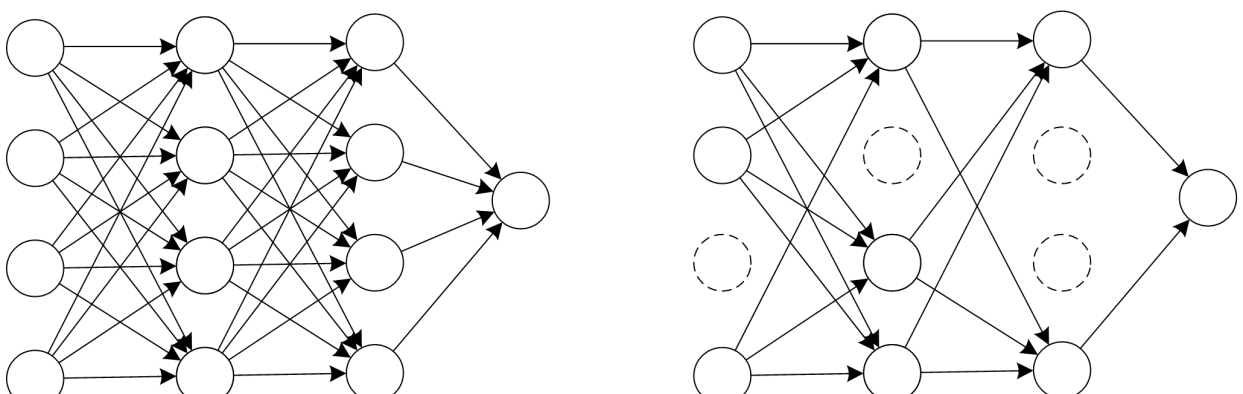
Regularization – Dropout

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.



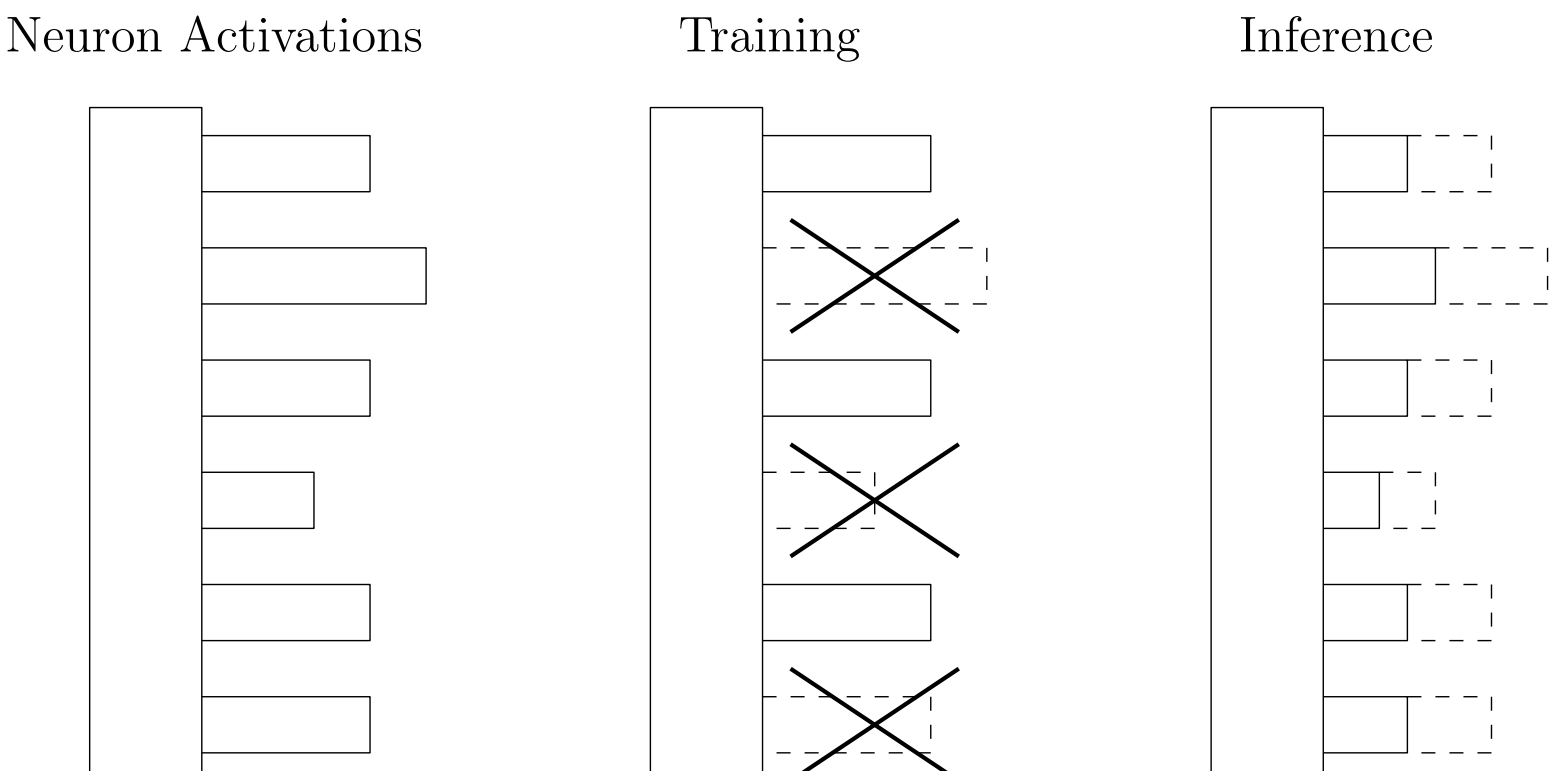
(a) Standard Neural Network

(b) Network after Dropout

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

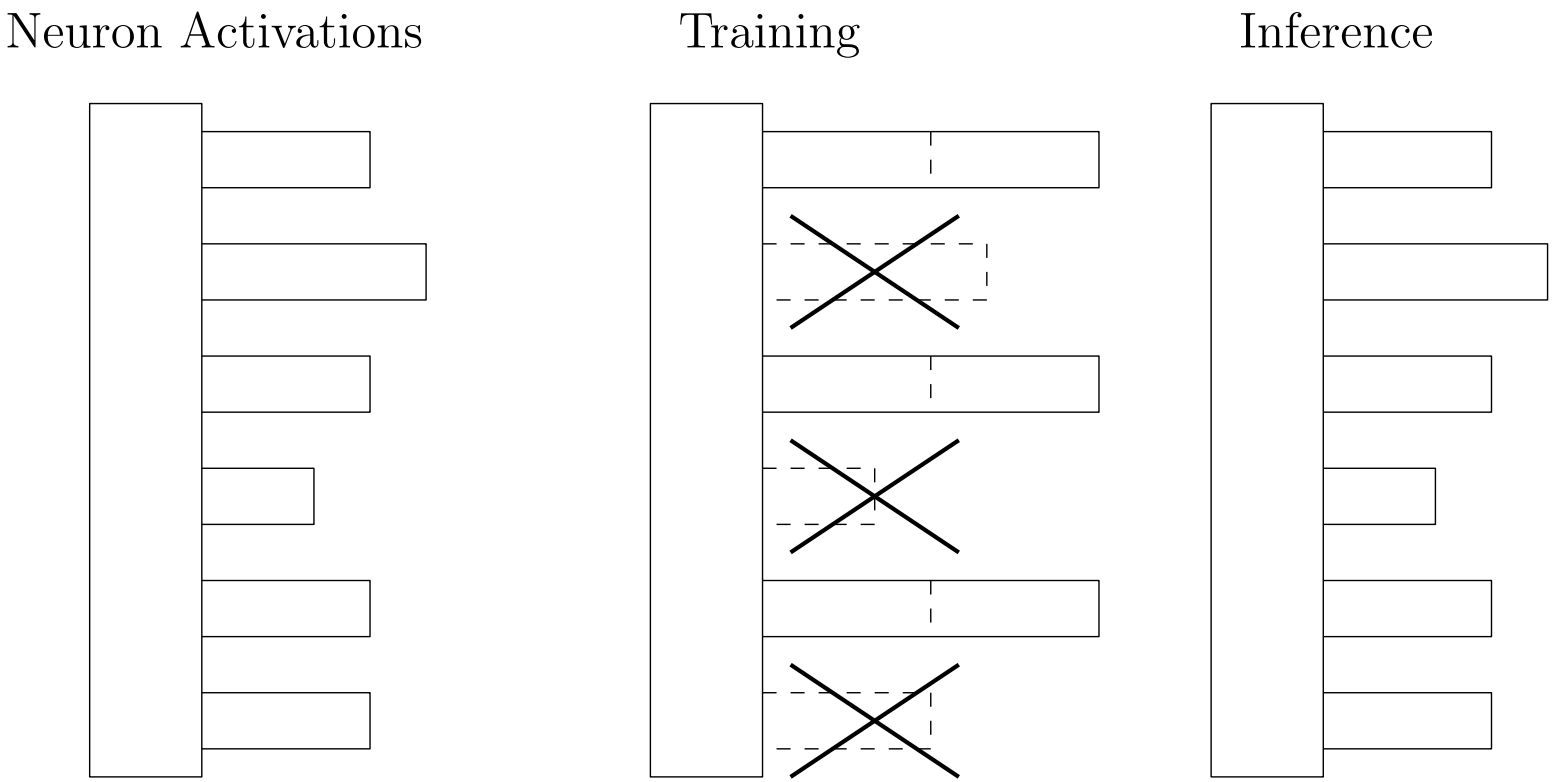
Regularization – Dropout

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.



Regularization – Dropout

Alternatively, we might *scale the activations up* during training by a factor of $1/(1-p)$.



Regularization – Dropout as Ensembling

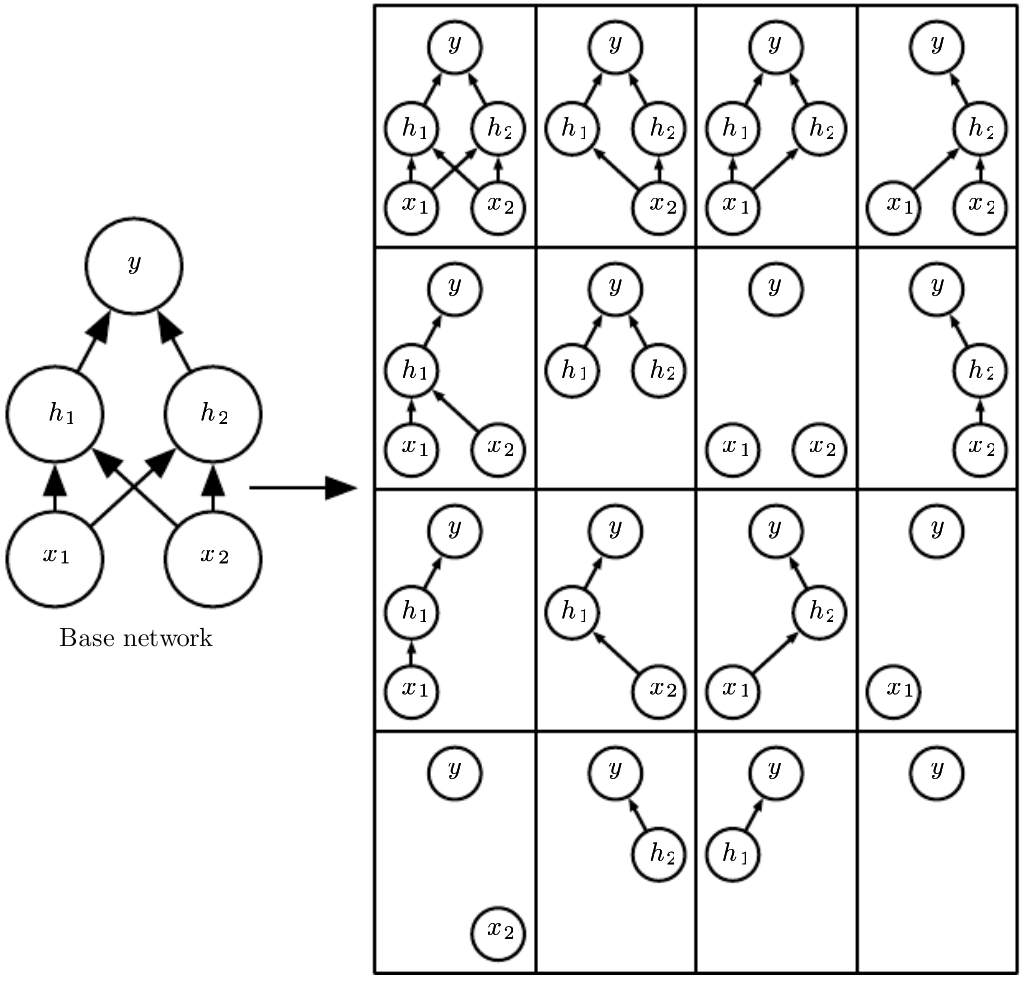
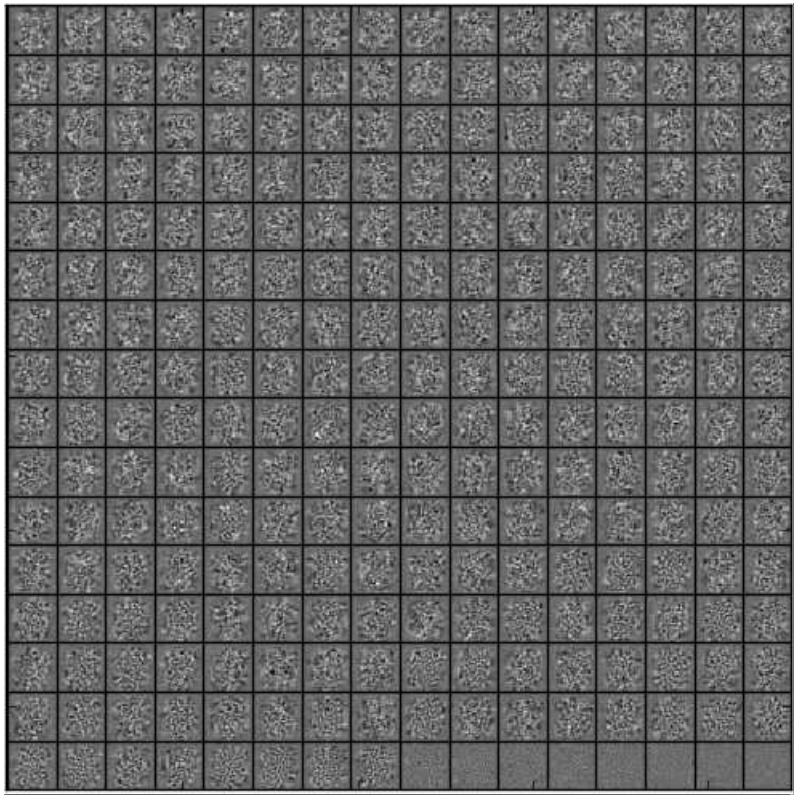
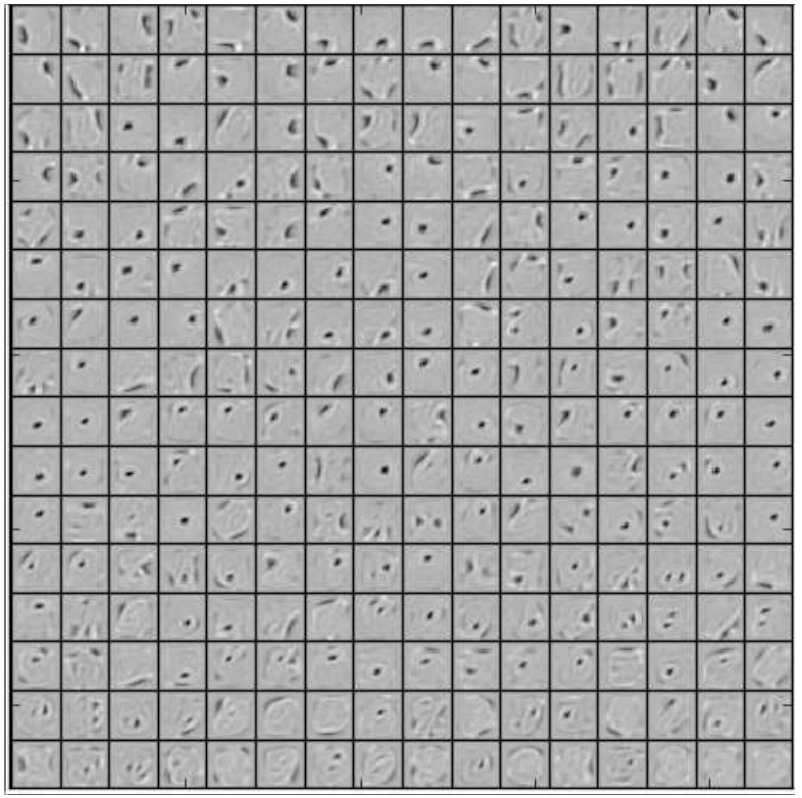


Figure 7.6, page 260 of Deep Learning Book, http://deeplearningbook.org

Regularization – Dropout Effect



(a) Without dropout



(b) Dropout with $p = 0.5$.

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

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

Regularization – Dropout Implementation

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

    def do_train():
        random_noise = tf.random.uniform(tf.shape(inputs))
        mask = tf.cast(tf.less(random_noise, rate), tf.float32)
        return inputs * mask / (1 - rate)

    if training == True:
        return do_train()
    if training == False:
        return do_inference()
    return tf.cond(training, do_train, do_inference)
```

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.

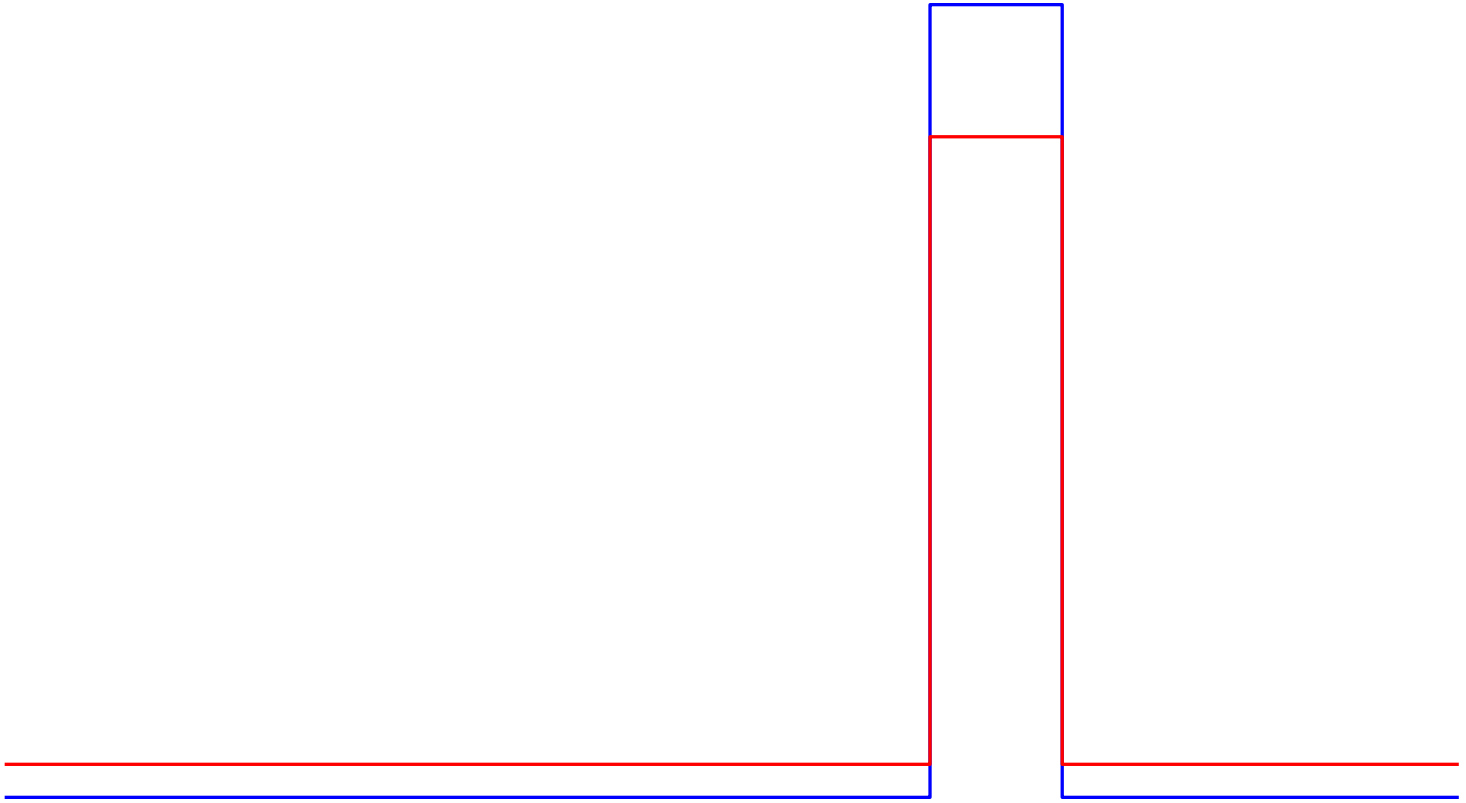
Ideally, we would like a full (non-sparse) categorical distribution of classes for training examples, but that is usually not available.

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

The target distribution is then

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

Regularization – Label Smoothing



Gold distribution

Smoothed distribution

Regularization – Good Defaults

When you need to regularize, then a good default strategy is to:

- use dropout on all hidden dense layers (not on the output layer), good default dropout rate is 0.5 (or use 0.3 if the model is underfitting);
- use L2 regularization for your 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.

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

- saturating non-linearities,
- parameter initialization strategies,
- gradient clipping.

Convergence – Saturating Non-linearities

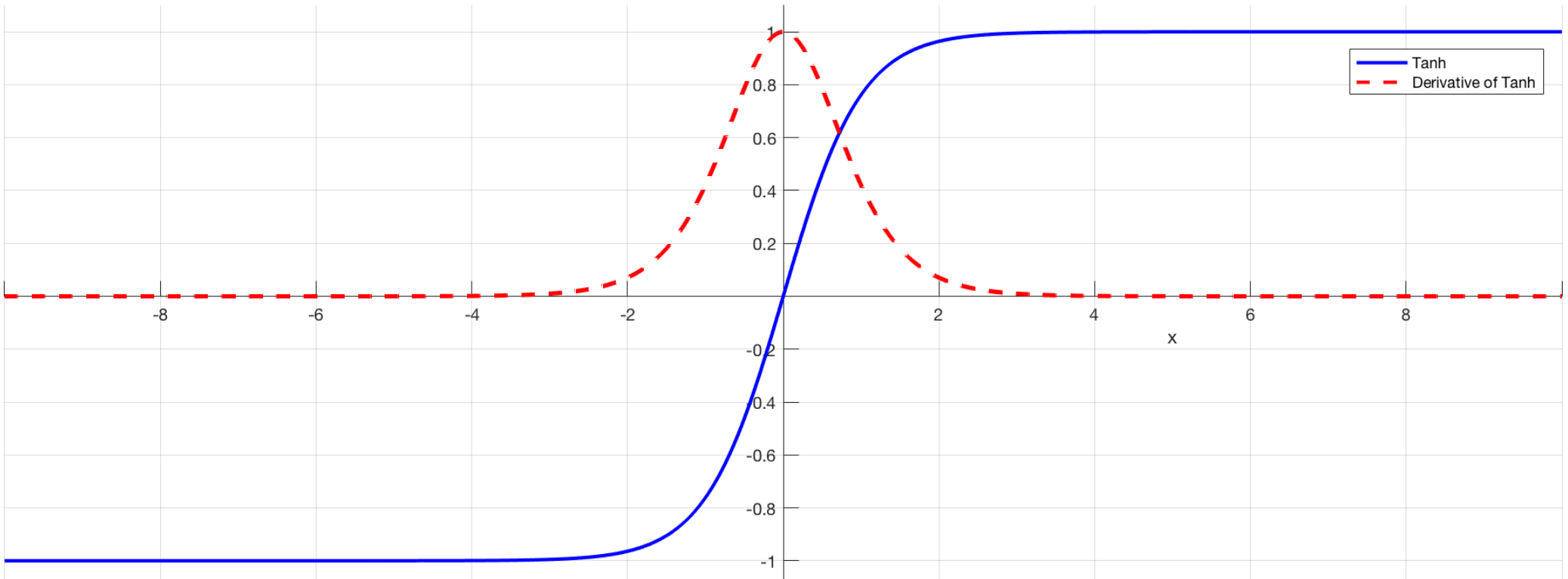


Image from https://isaacchanghau.github.io/images/deeplearning/activationfunction/tanh.png.

Convergence – Parameter Initialization

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

- Biases are usually initialized to a constant value, usually 0.
- Weights are usually initialized to small random values, either with uniform or normal distribution.
  - The scale matters for deep networks!
  - Originally, people used $U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$ distribution.
  - 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

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

Convergence – Parameter Initialization

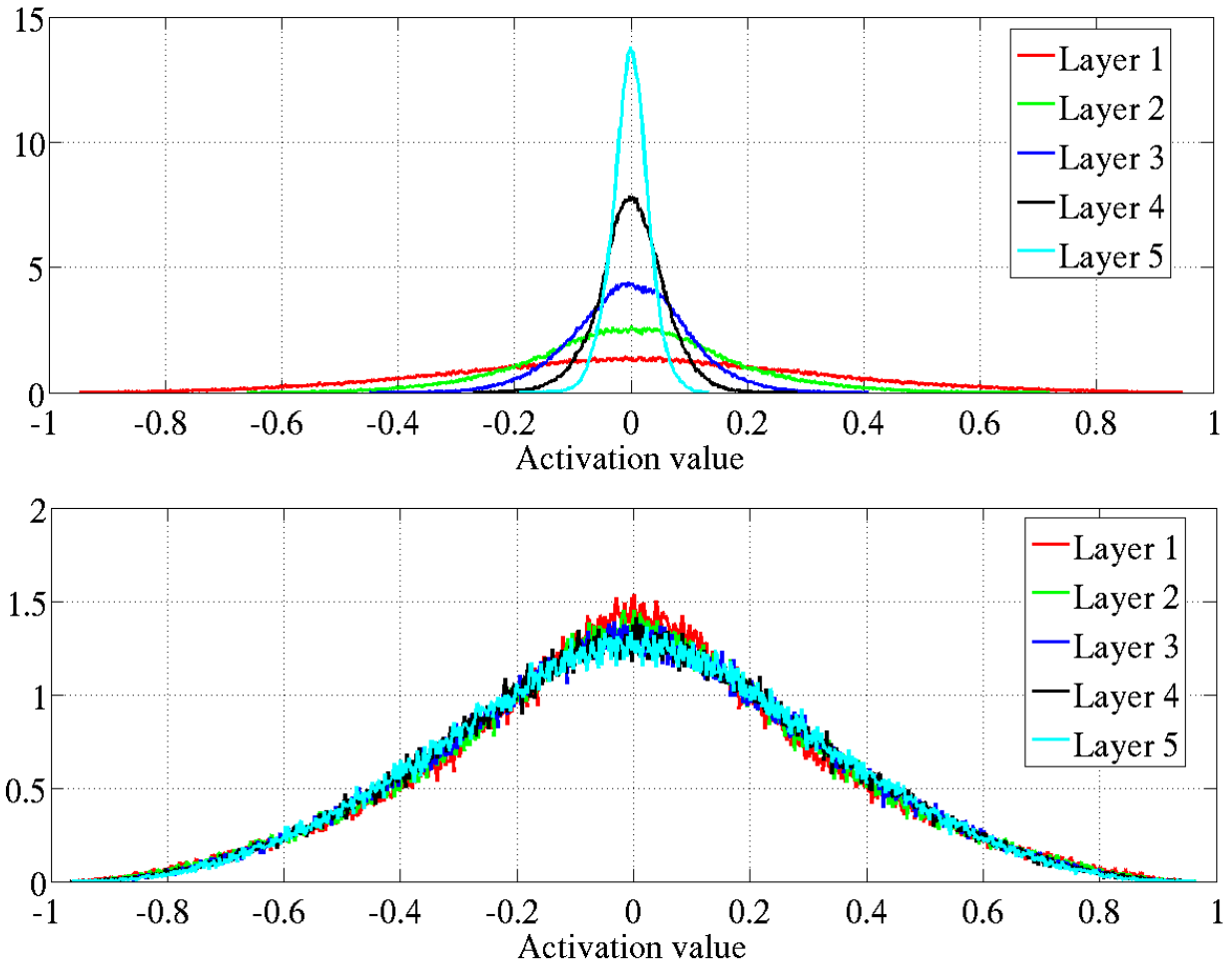


Figure 6 of paper "Understanding the difficulty of training deep feedforward neural networks", http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

Convergence – Parameter Initialization

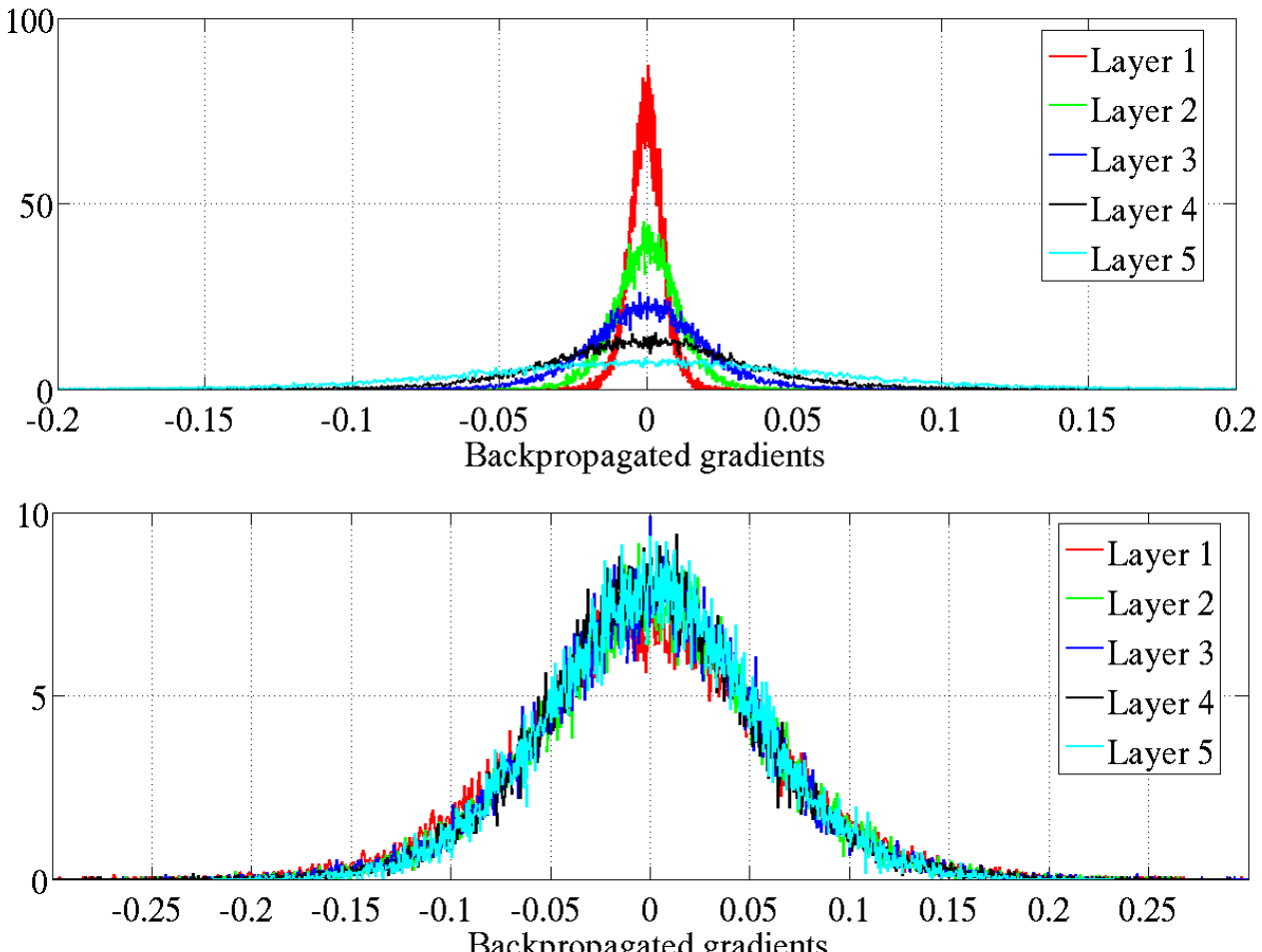


Figure 7 of paper "Understanding the difficulty of training deep feedforward neural networks", http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

Convergence – Gradient Clipping

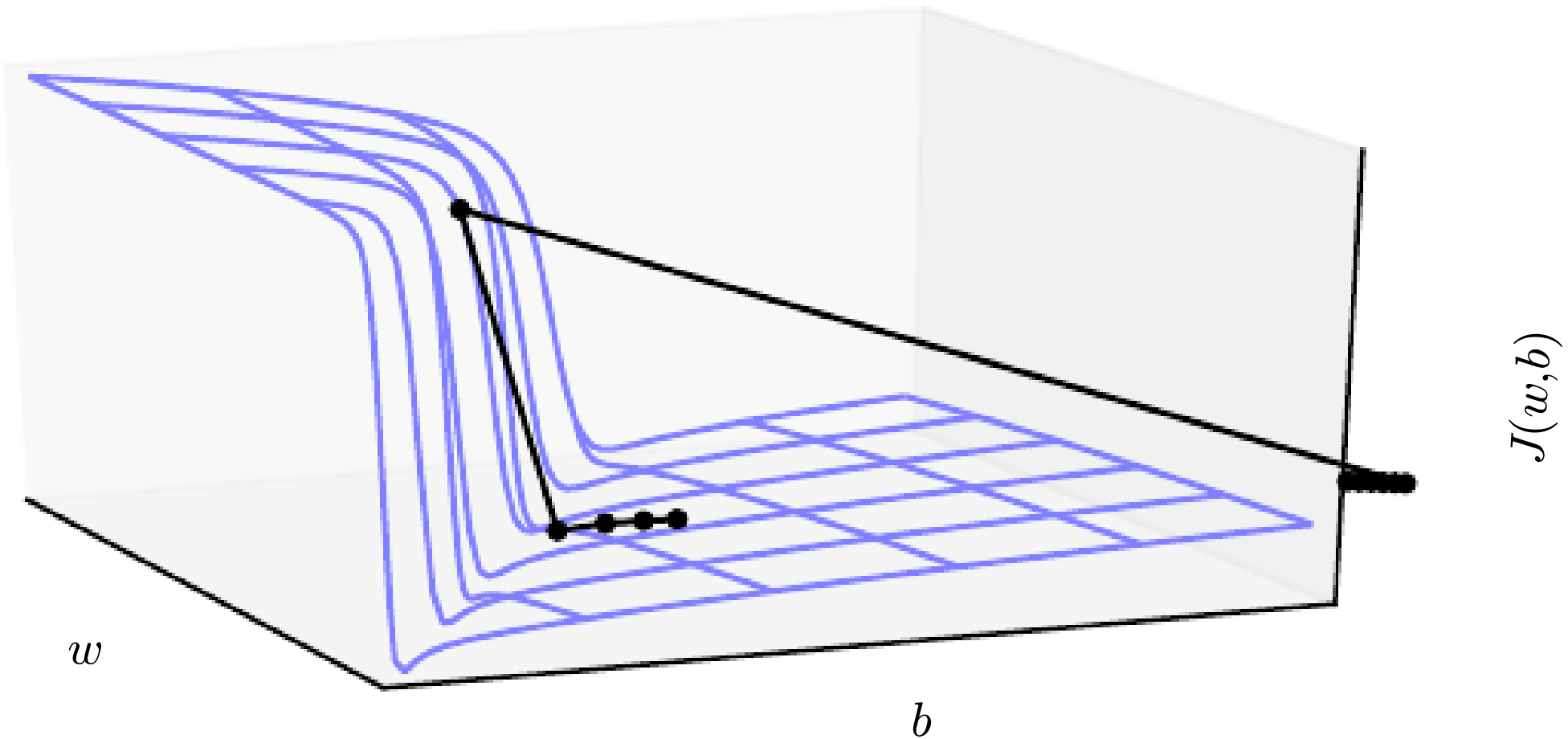


Figure 8.3, page 289 of Deep Learning Book, http://deeplearningbook.org

Convergence – Gradient Clipping

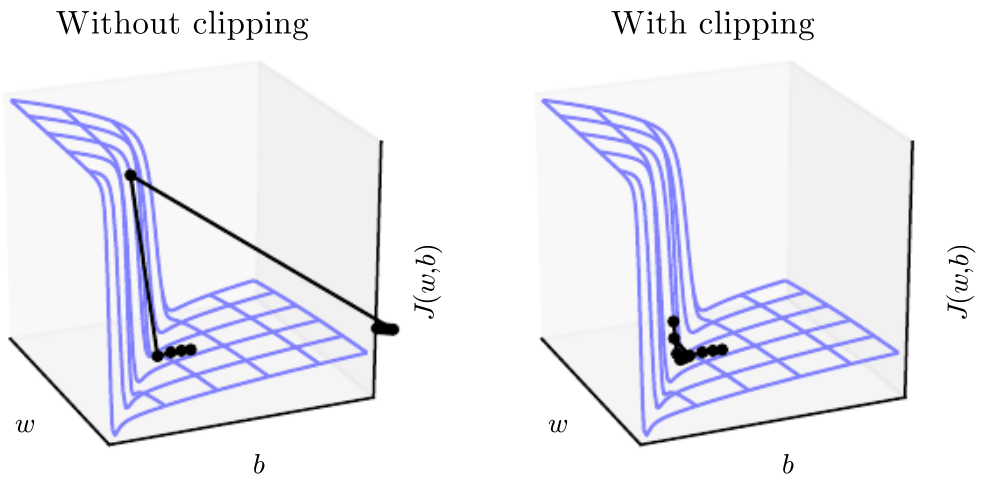


Figure 10.17, page 414 of Deep Learning Book, http://deeplearningbook.org

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

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

The clipping can be per weight (`clipvalue` of `tf.optimizers.Optimizer`), per variable or for the gradient as a whole (`clipnorm` of `tf.optimizers.Optimizer`).