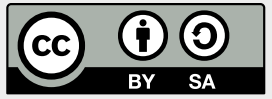
Training Neural Networks II

Milan Straka

February 28, 2022

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

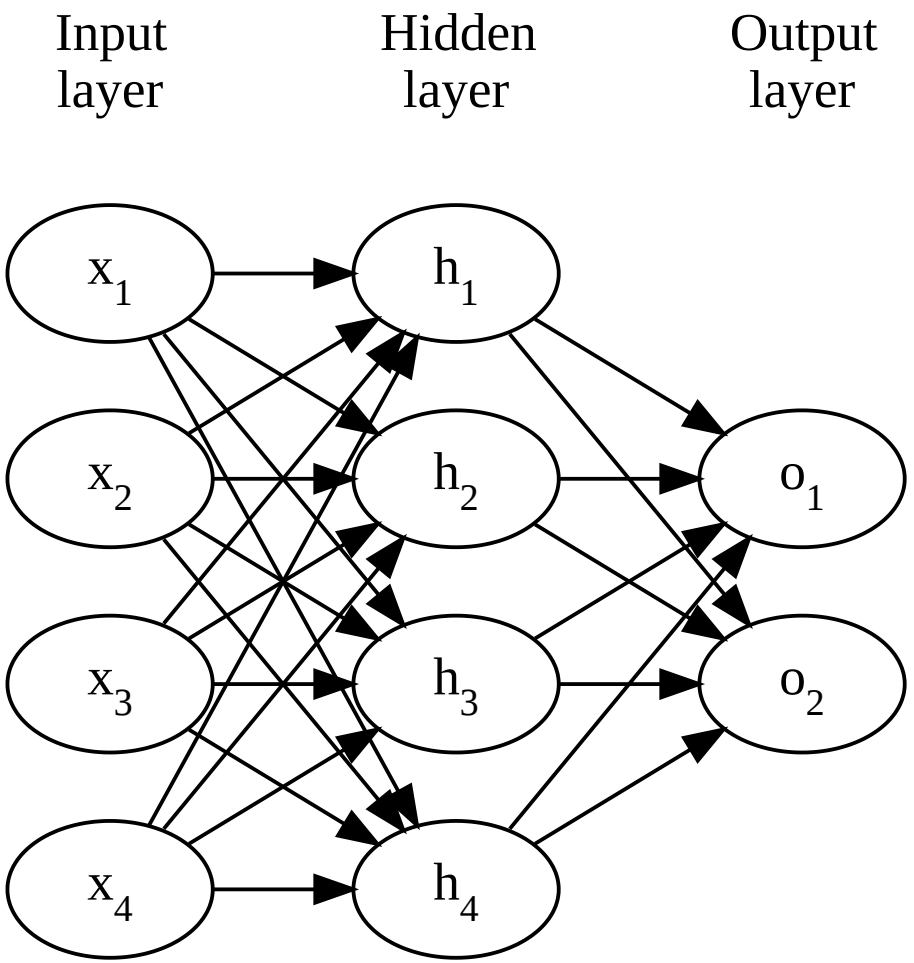
Let us have a dataset with training, validation and test sets, each containing examples $(\boldsymbol{x}, y)$. Depending on y, consider one of the following output activation functions:

$$\begin{cases} \text{none} & \text{if } y \in \mathbb{R}, \\ \sigma & \text{if } y \text{ is a probability of an outcome}, \\ \text{softmax} & \text{if } y \text{ is a gold class index out of } K \text{ classes (or a full distribution)}. \end{cases}$$

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

BTW, there are of course many functions, which could be used as output activations instead of σ *and* $\mathrm{softmax}$*; however,* σ *and* $\mathrm{softmax}$ *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



We have

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

where

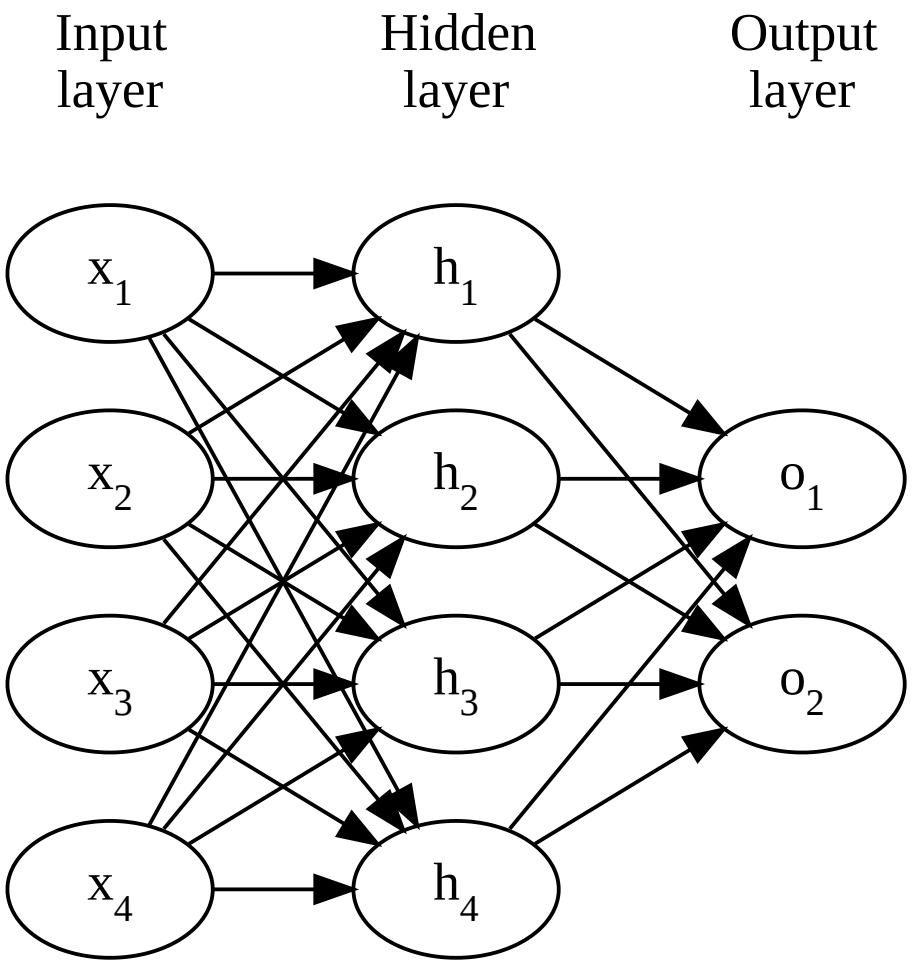
- $\boldsymbol{W}^{(1)} \in \mathbb{R}^{D \times H}$ is a matrix of **weights**,
- $\boldsymbol{b}^{(1)} \in \mathbb{R}^{H}$ is a vector of **biases**,
- $f^{(1)}$ is an activation function.

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

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

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

Putting It All Together



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

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 of its adaptive variants), updating the parameters to minimize the loss:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial L}{\partial \theta_i}, \quad \text{or in vector notation,} \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial L}{\partial \boldsymbol{\theta}}.$$

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.

Practical Issues

- Processing all data in **batches**, as a matrix $\boldsymbol{X}$ with rows of batch examples.
- Vector representation of the network.

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

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

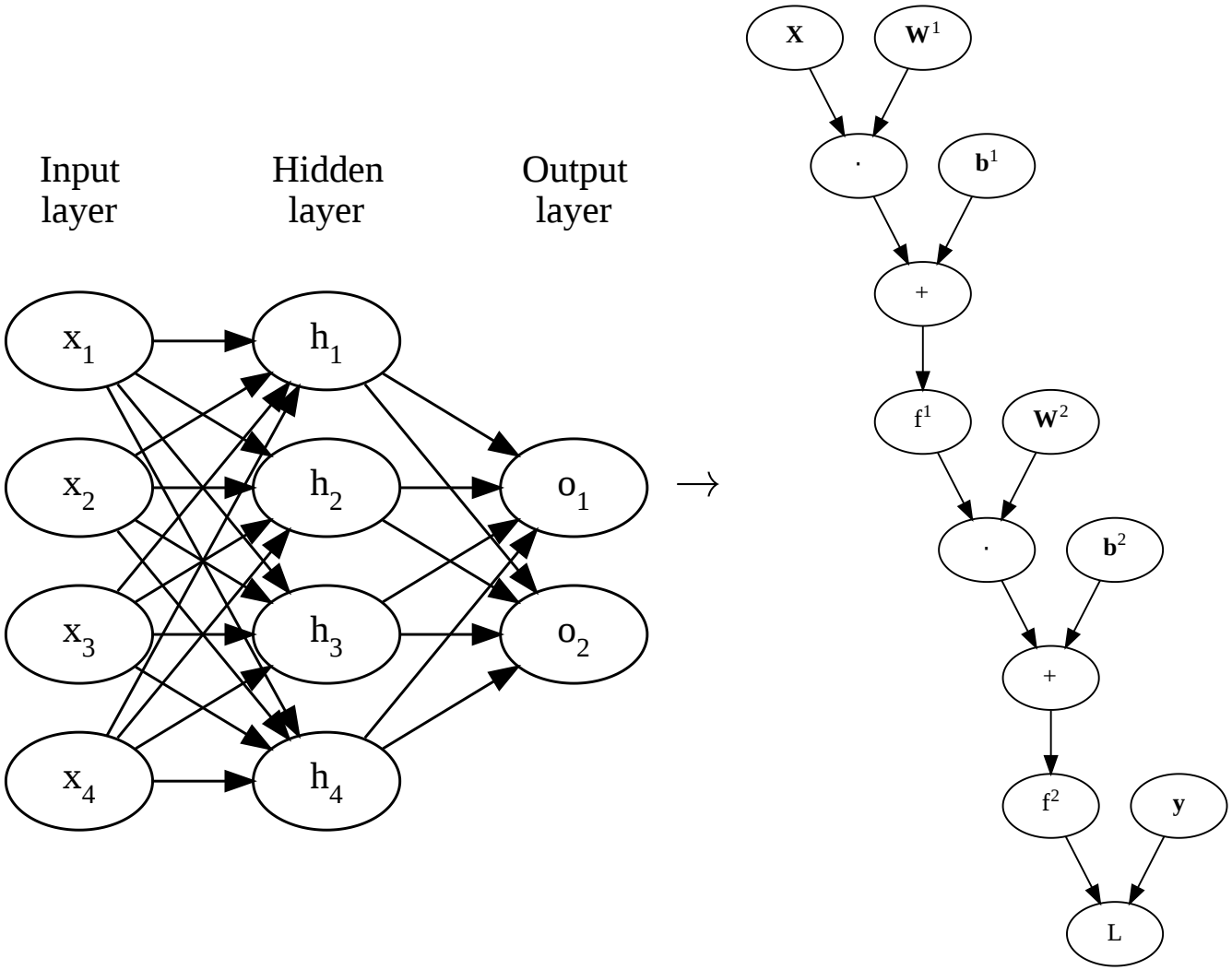
$$\boldsymbol{O} = f^{(2)}\left(\boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right) = f^{(2)}\left(f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right)$$

  The derivatives

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

  are then batches of 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 (SiLU), Mish, ...
Output function	none, σ	none, σ, $\mathrm{softmax}$
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 TensorFlow, the losses and metrics are available in `tf.losses` and `tf.metrics` (aliases for `tf.keras.losses` and `tf.keras.metrics`).

TF Losses

The `tf.losses` offer two sets of APIs. The current ones are loss classes like

```
tf.losses.MeanSquaredError(
    reduction=tf.losses.Reduction.AUTO, name='mean_squared_error'
)
```

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

```
__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.

- `tf.losses.Reduction.SUM_OVER_BATCH_SIZE`, which is the default of `.AUTO`;
- `tf.losses.Reduction.SUM`;
- `tf.losses.Reduction.NONE`.

TF Cross-entropy Losses

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

- `tf.losses.BinaryCrossentropy`: the gold and predicted distributions are Bernoulli distributions (i.e., a single probability);
- `tf.losses.CategoricalCrossentropy`: the gold and predicted distributions are categorical distributions;
- `tf.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.

Old losses API

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

TF 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 `tf.losses.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

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

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

TF Metrics

Apart from analogues of the losses

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

the `tf.metrics` module provides

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

Derivative of MSE Loss

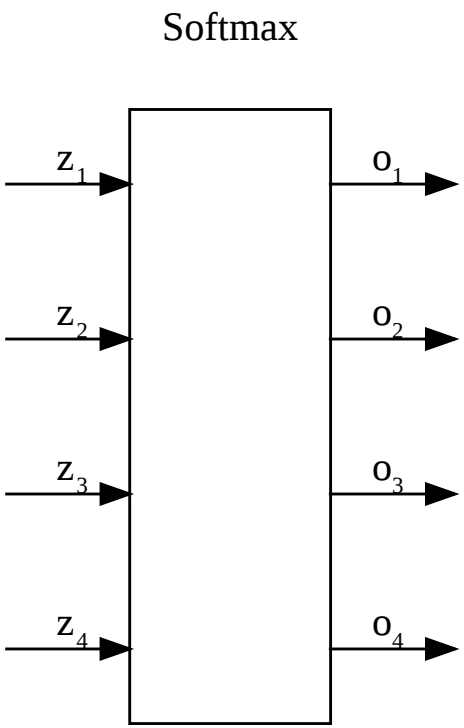
Given the MSE loss of

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

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

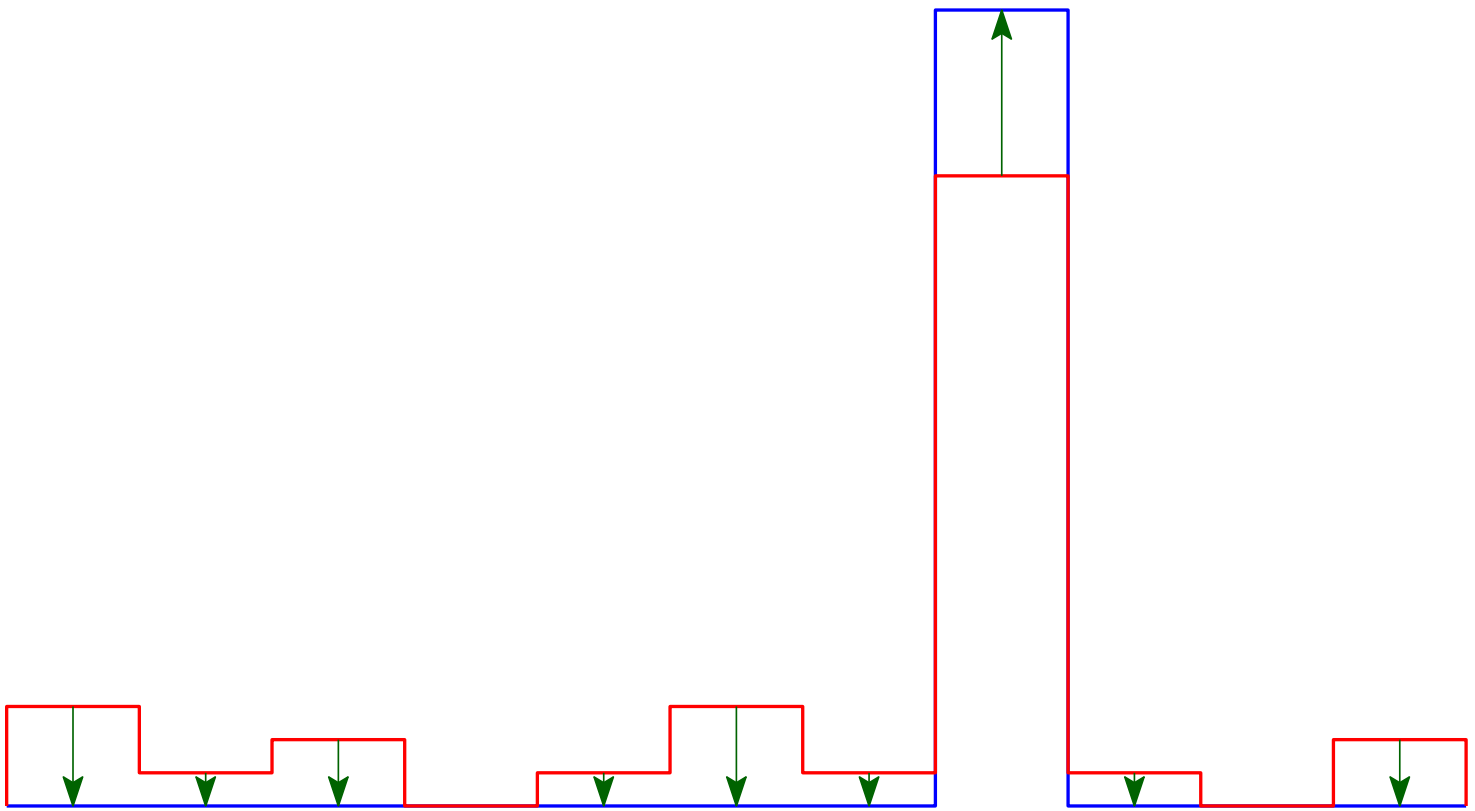
$$L(\text{softmax}(\boldsymbol{z}), gold) = -\log o_{gold}.$$

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

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

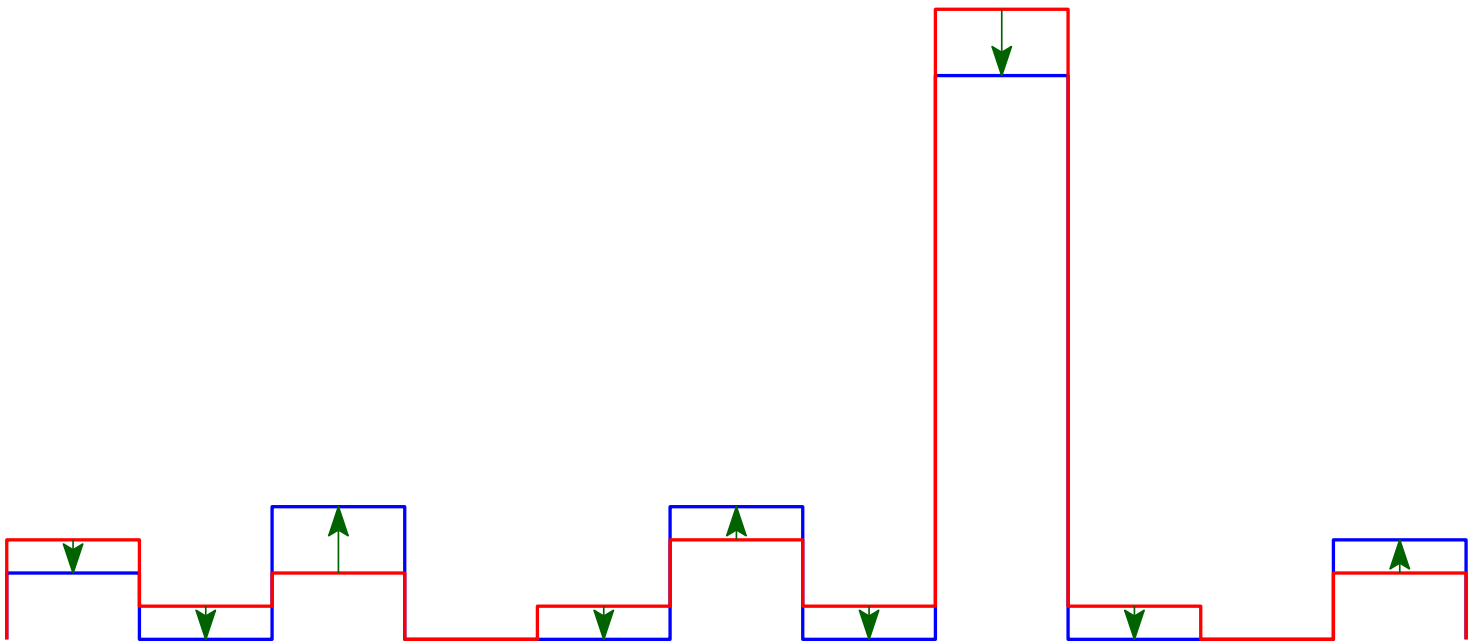
Repeating the previous procedure for each target probability, we obtain

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

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
- L^2, L^1 regularization
- Dataset augmentation
- Ensembling
- Dropout
- Label smoothing

Regularization – Early Stopping

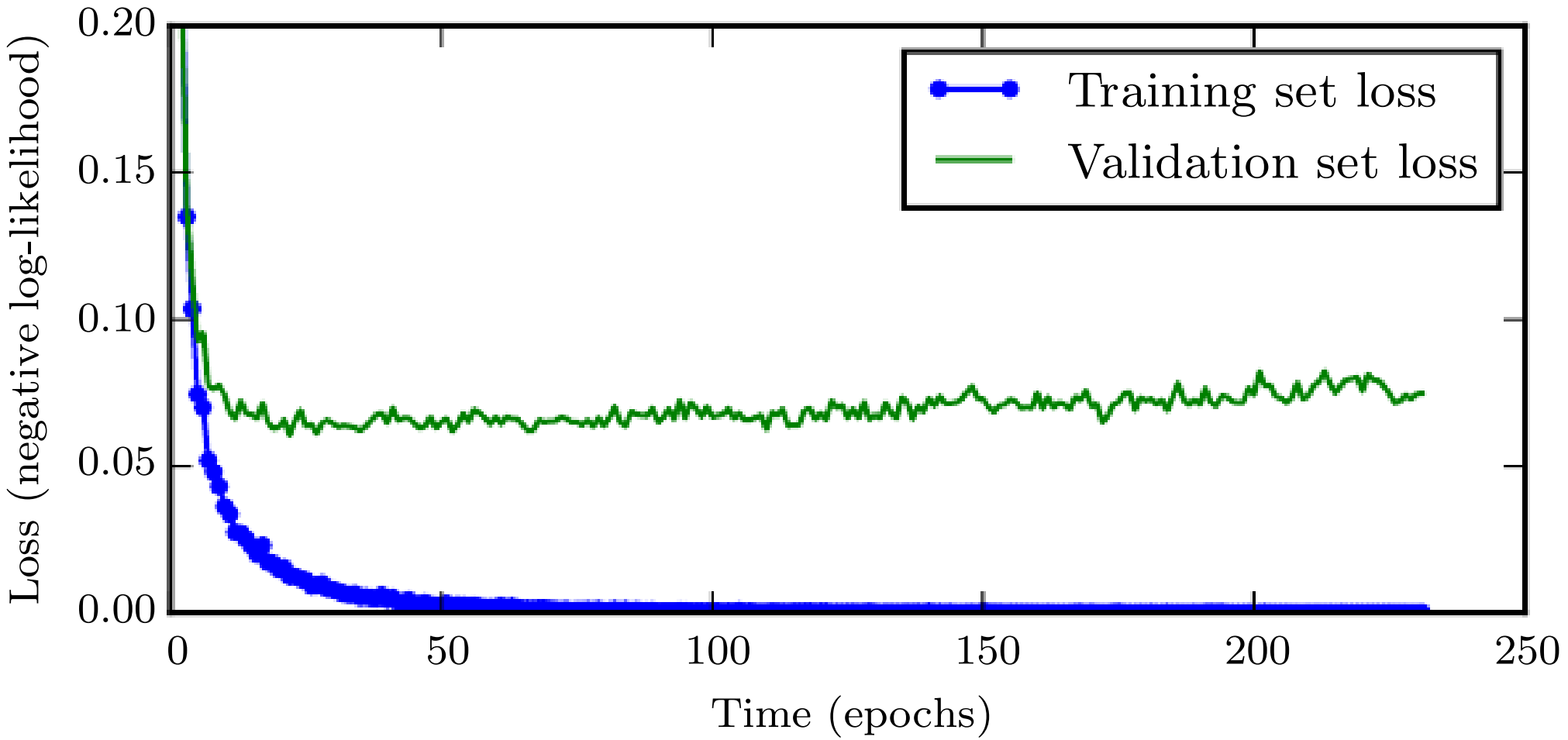


Figure 7.3 of "Deep Learning" book, https://www.deeplearningbook.org

L2 Regularization

We prefer models with parameters small under L^2 metric.

The L^2 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

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial J}{\partial \theta_i} - 2\alpha\lambda\theta_i, \text{ or in vector notation, } \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial L}{\partial \boldsymbol{\theta}} - 2\alpha\lambda\boldsymbol{\theta}.$$

This can be also written as

$$\theta_i \leftarrow \theta_i(1 - 2\alpha\lambda) - \alpha \frac{\partial J}{\partial \theta_i}, \text{ or in vector notation, } \boldsymbol{\theta} \leftarrow \boldsymbol{\theta}(1 - 2\alpha\lambda) - \alpha \frac{\partial L}{\partial \boldsymbol{\theta}}.$$

L2 Regularization

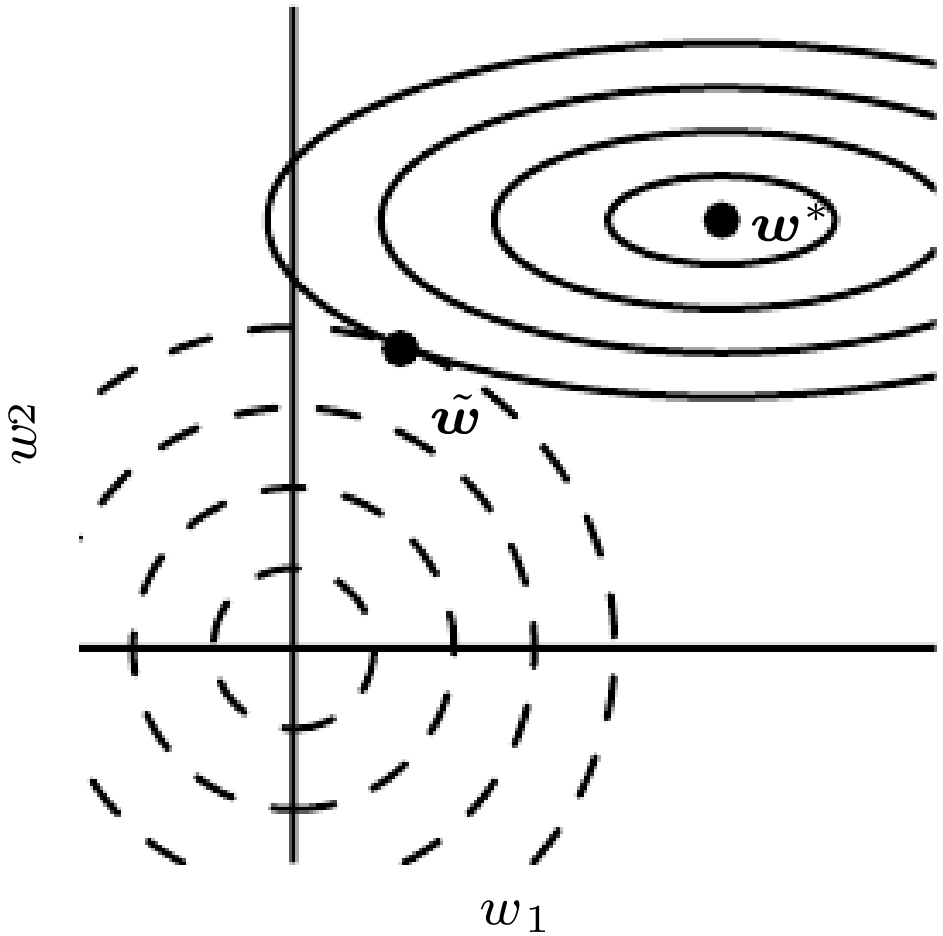


Figure 7.1 of "Deep Learning" book, https://www.deeplearningbook.org

L2 Regularization as MAP

Another way to arrive at L^2 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 stating that

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

we can rewrite the MAP estimate to

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

L2 Regularization as MAP

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

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

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

By substituting the probability of the Gaussian prior, we get

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

L1 Regularization

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

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

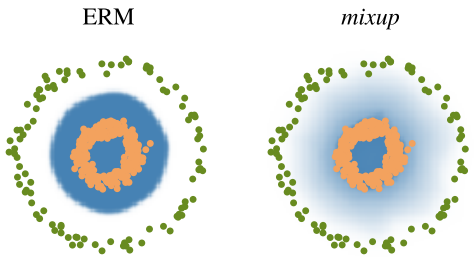
The corresponding SGD update is then

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial J}{\partial \theta_i} - \min\left(\alpha\lambda, |\theta_i|\right) \operatorname{sign}(\theta_i).$$

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 "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. If we denote the prediction of a model y_i 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 $\mathbb{E}\left[(y_i(\boldsymbol{x}) - y)^2\right] = \mathbb{E}\left[\varepsilon_i^2(\boldsymbol{x})\right]$.

Because for uncorrelated identically distributed random values x_i we have

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

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

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).

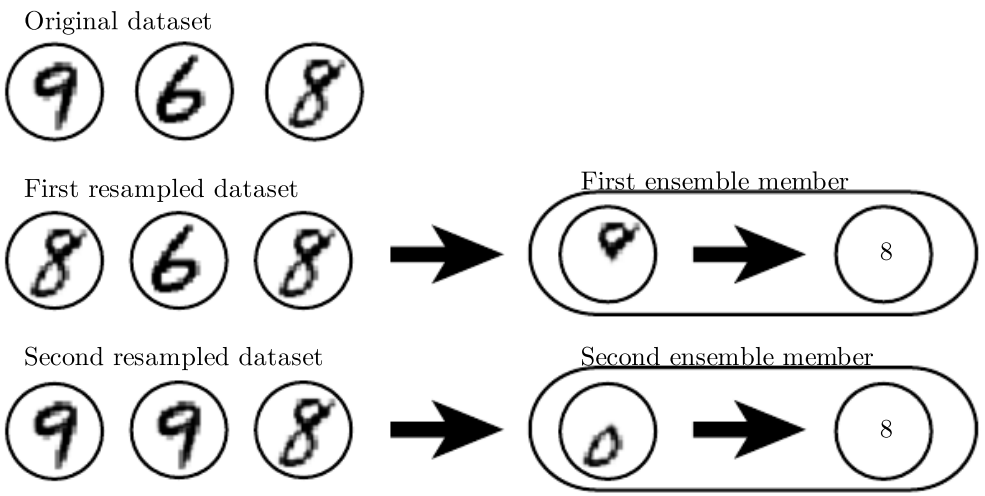


Figure 7.5 of "Deep Learning" book, https://www.deeplearningbook.org

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

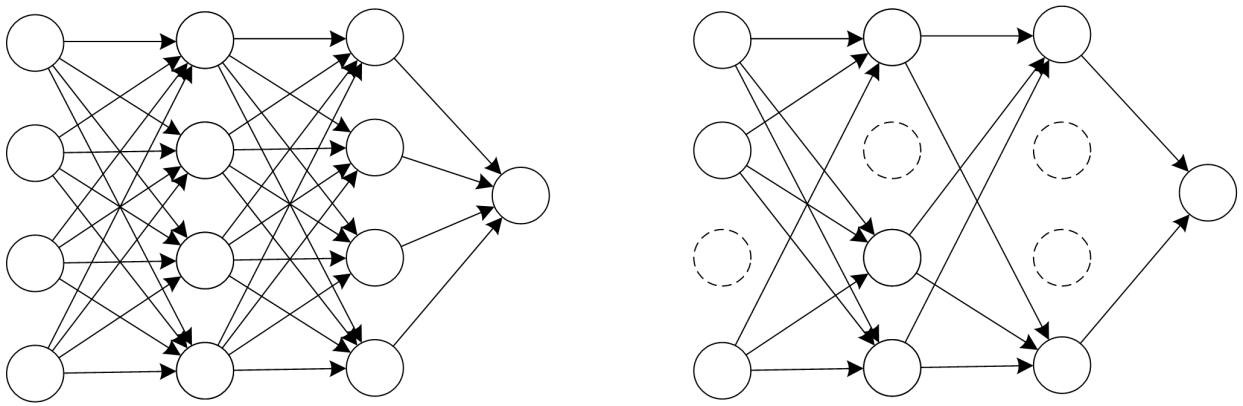
Regularization – Dropout

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.



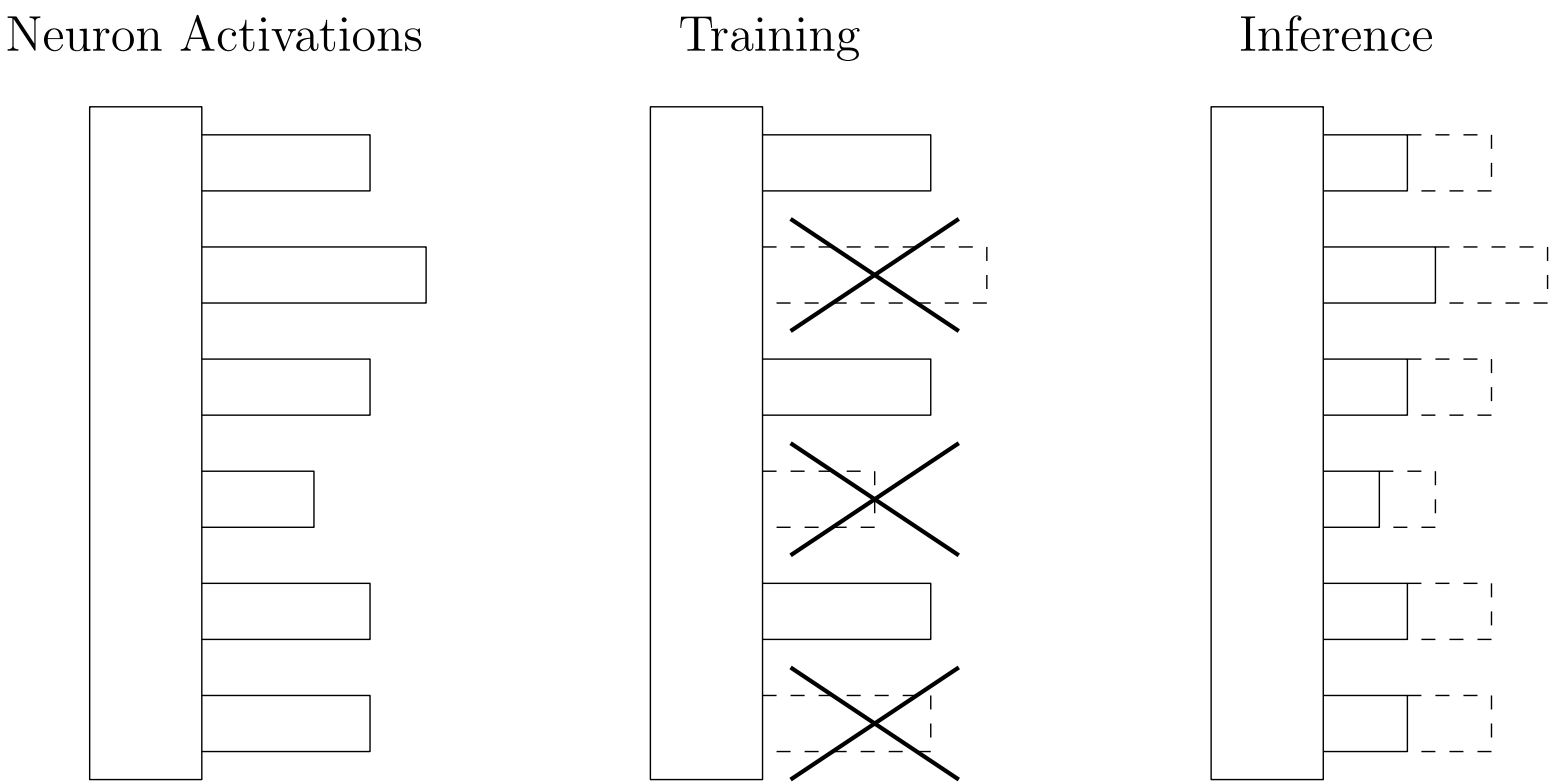
(a) Standard Neural Network

(b) Network after Dropout

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

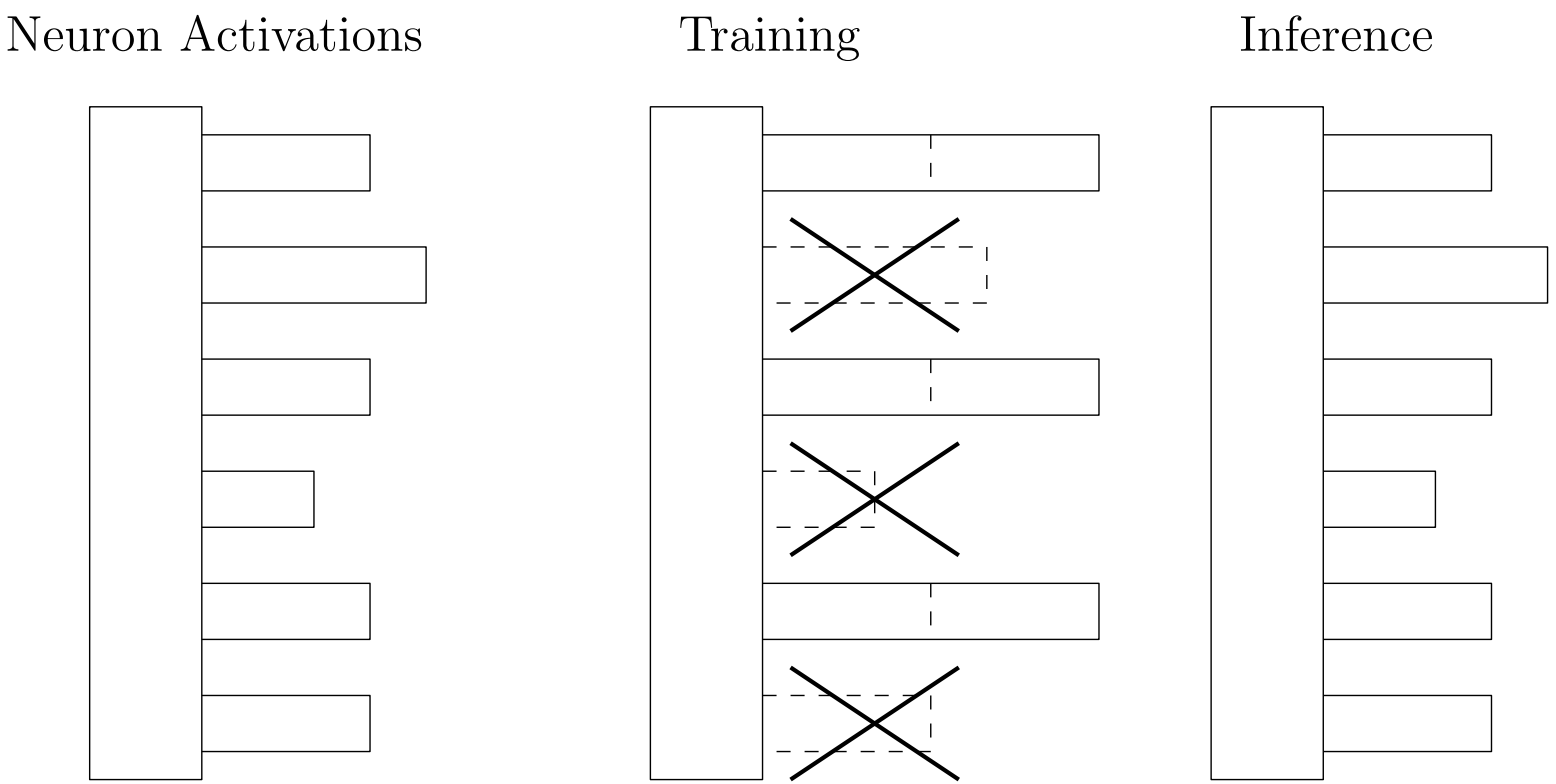
Regularization – Dropout

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

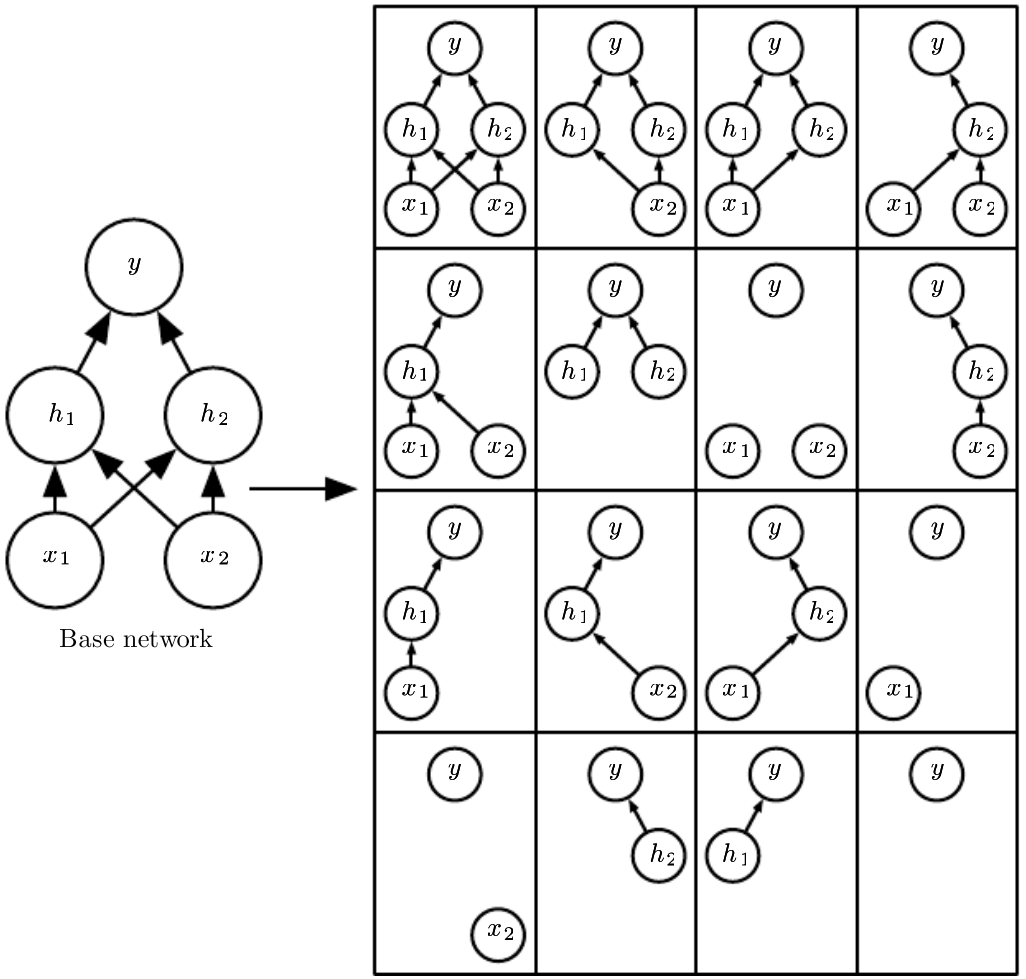


Figure 7.6 of "Deep Learning" book, https://www.deeplearningbook.org

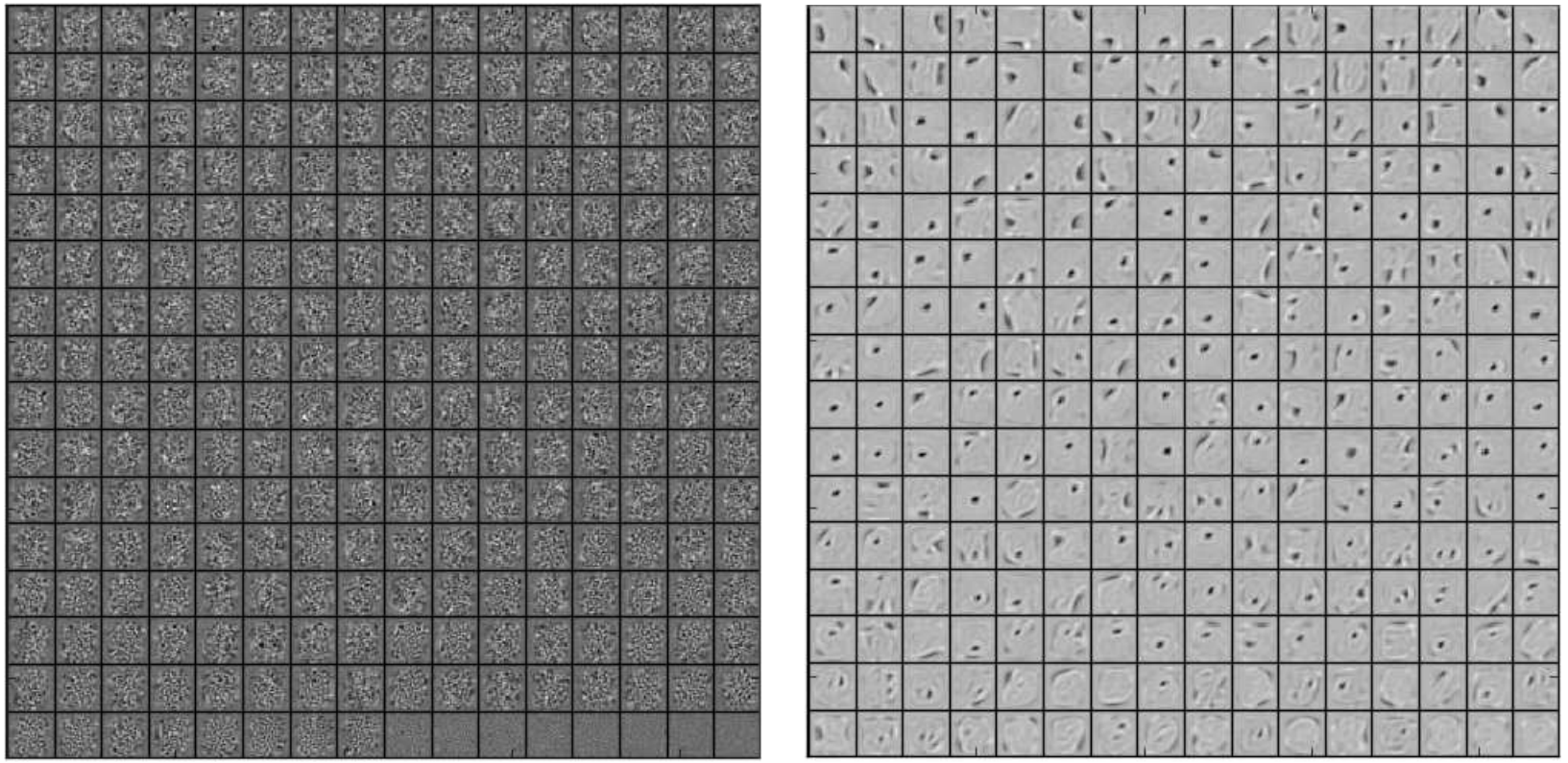
Regularization – Dropout Implementation

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

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

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

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 "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.

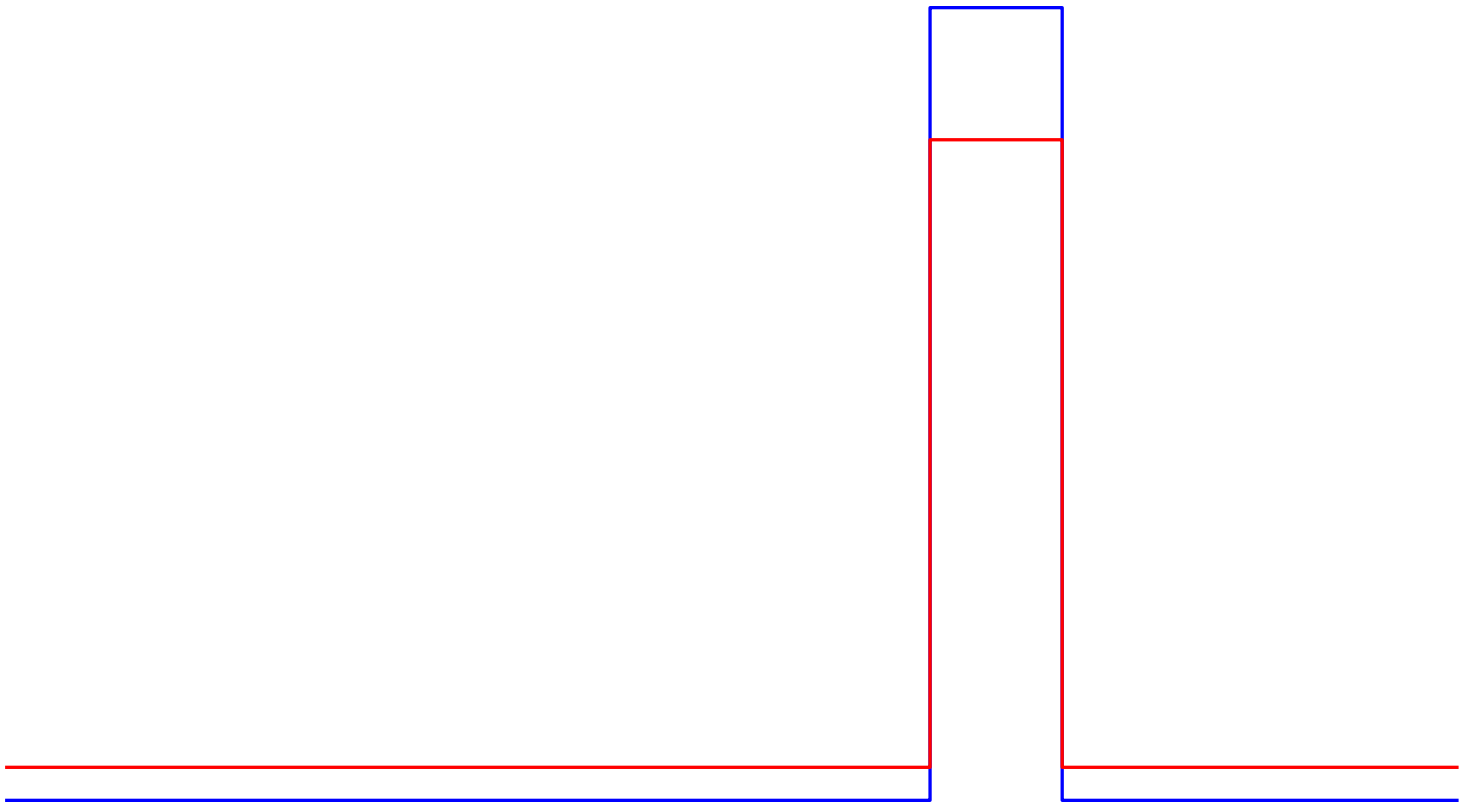
Ideally, we would like a full (non-sparse) categorical distribution of classes for training examples, but that is usually not available.

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

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

Regularization – Label Smoothing



Gold distribution

Smoothed distribution

Regularization – Good Defaults

When you need to regularize (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 if the model is underfitting);
- use L^2 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

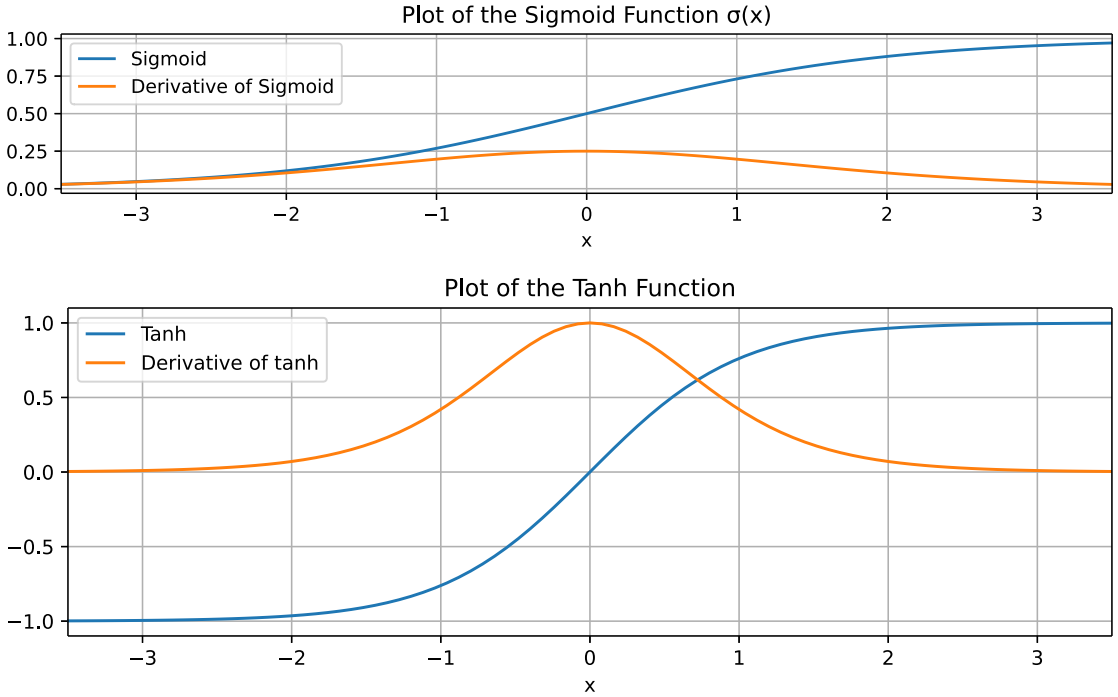
The training process might or might not converge. Even if it does, it might converge slowly or quickly.

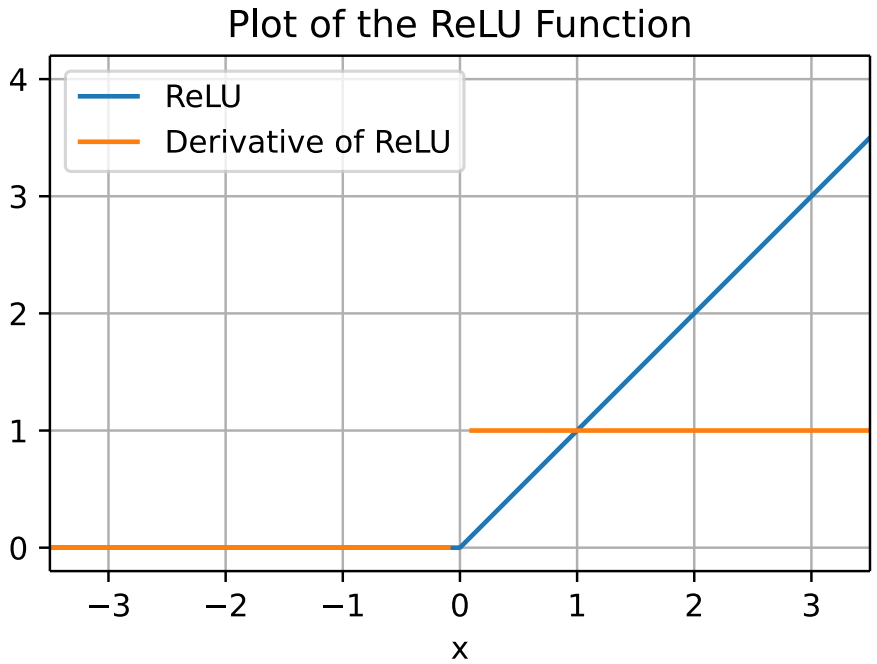
A major issue of convergence of deep networks is to make sure that the gradient with respect to all parameters is reasonable at all times, i.e., it does not decrease or increase too much with depth or in different batches.

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





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

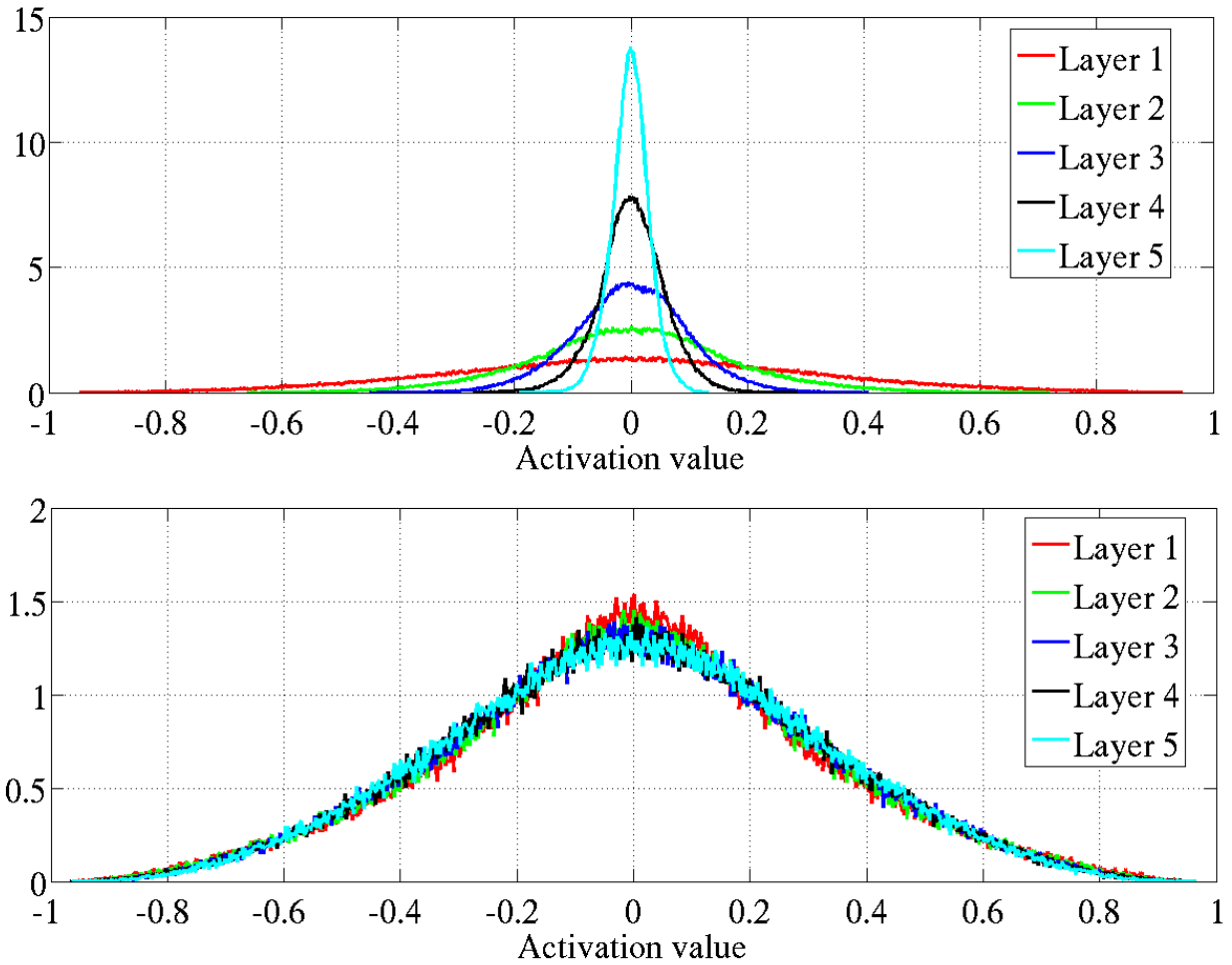


Figure 6 of "Understanding the difficulty of training deep feedforward neural networks", http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Convergence – Parameter Initialization

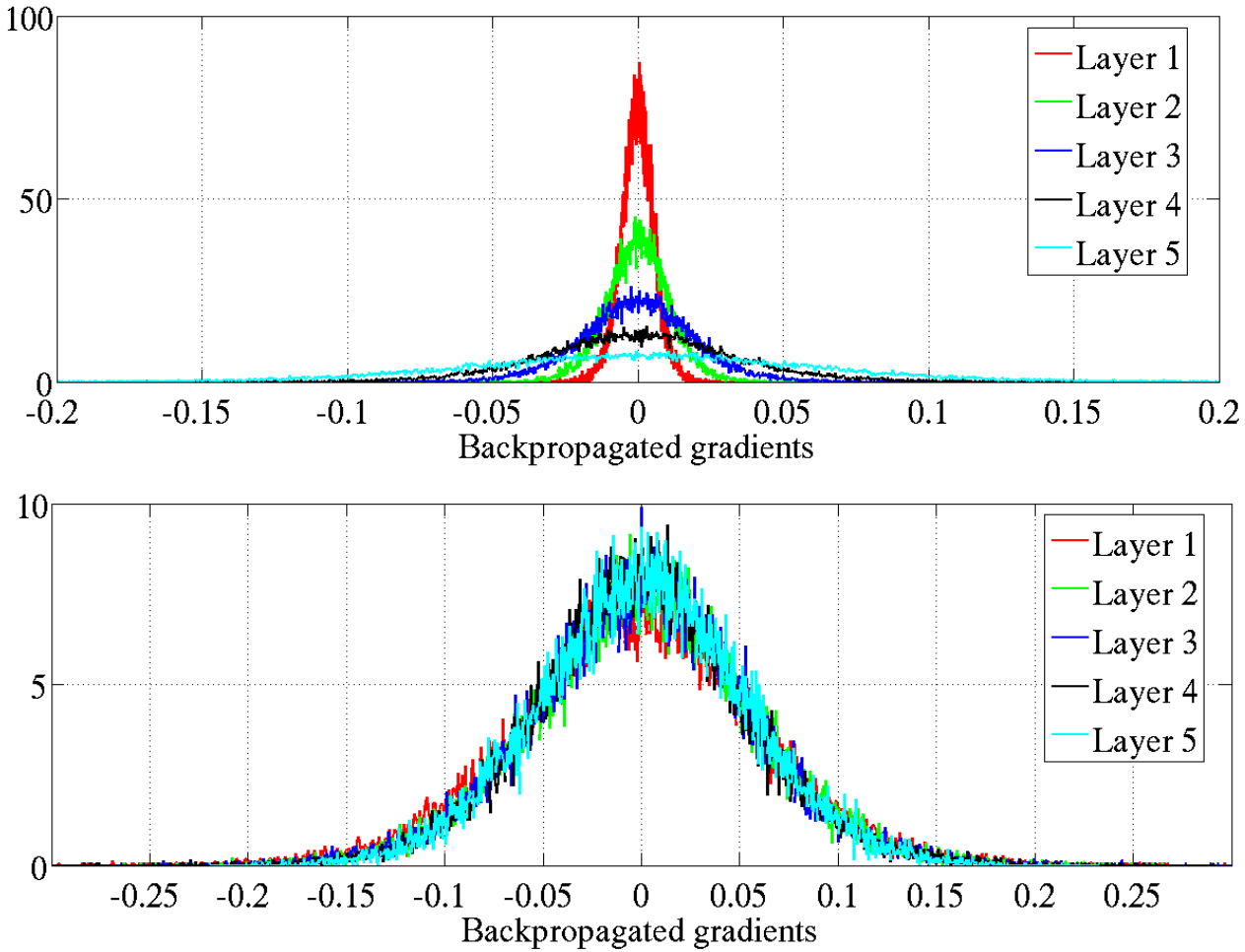


Figure 7 of "Understanding the difficulty of training deep feedforward neural networks", http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Convergence – Gradient Clipping

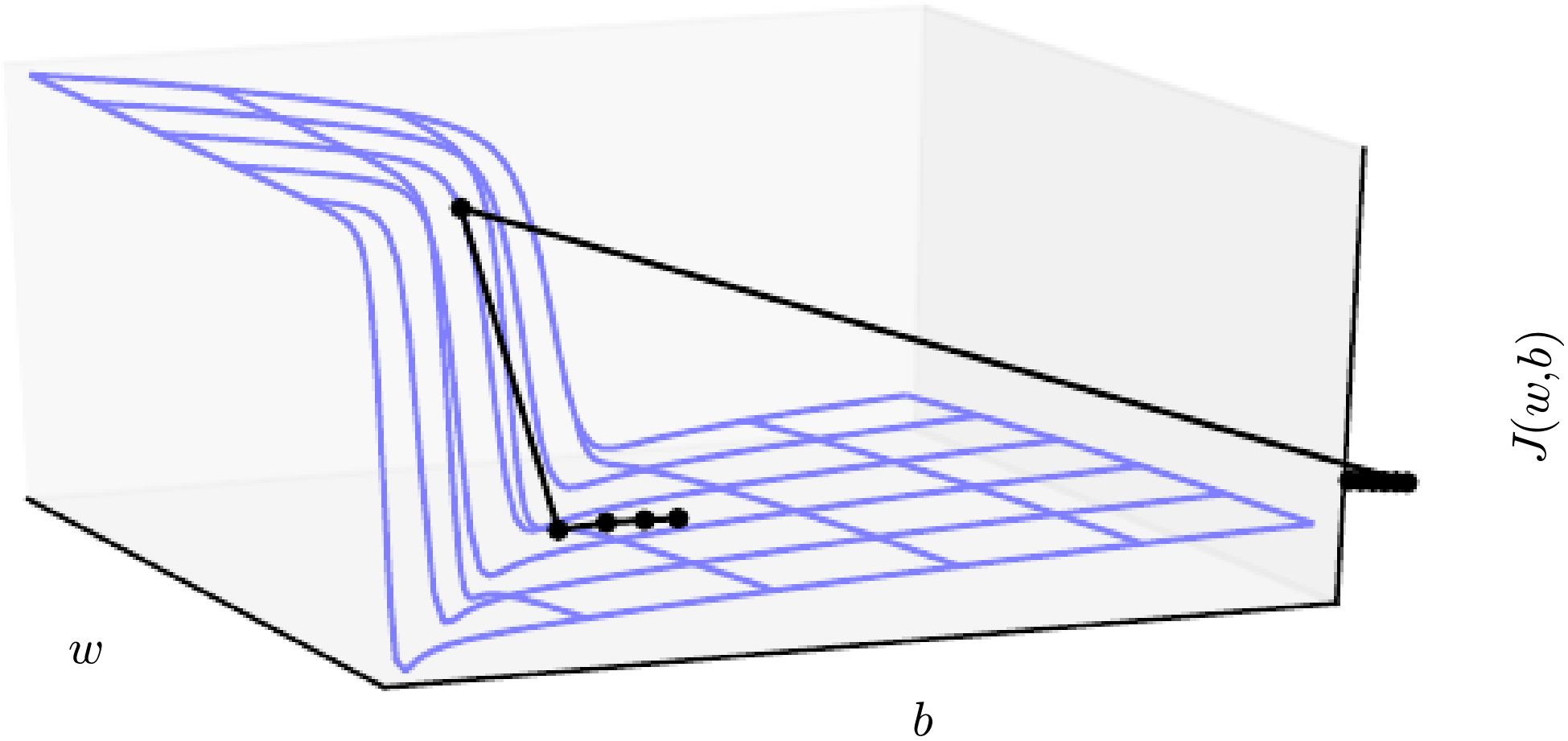


Figure 8.3 of "Deep Learning" book, https://www.deeplearningbook.org

Convergence – Gradient Clipping

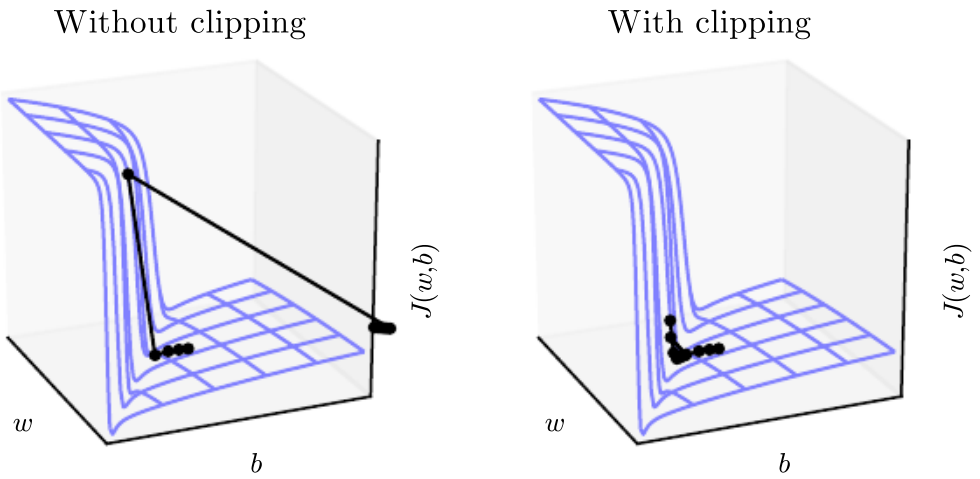


Figure 10.17 of "Deep Learning" book, https://www.deeplearningbook.org

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

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

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