Training Neural Networks II

Milan Straka

March 4, 2025

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

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

CC BY SA
unless otherwise stated

Neural Network Training Summary

Putting It All Together

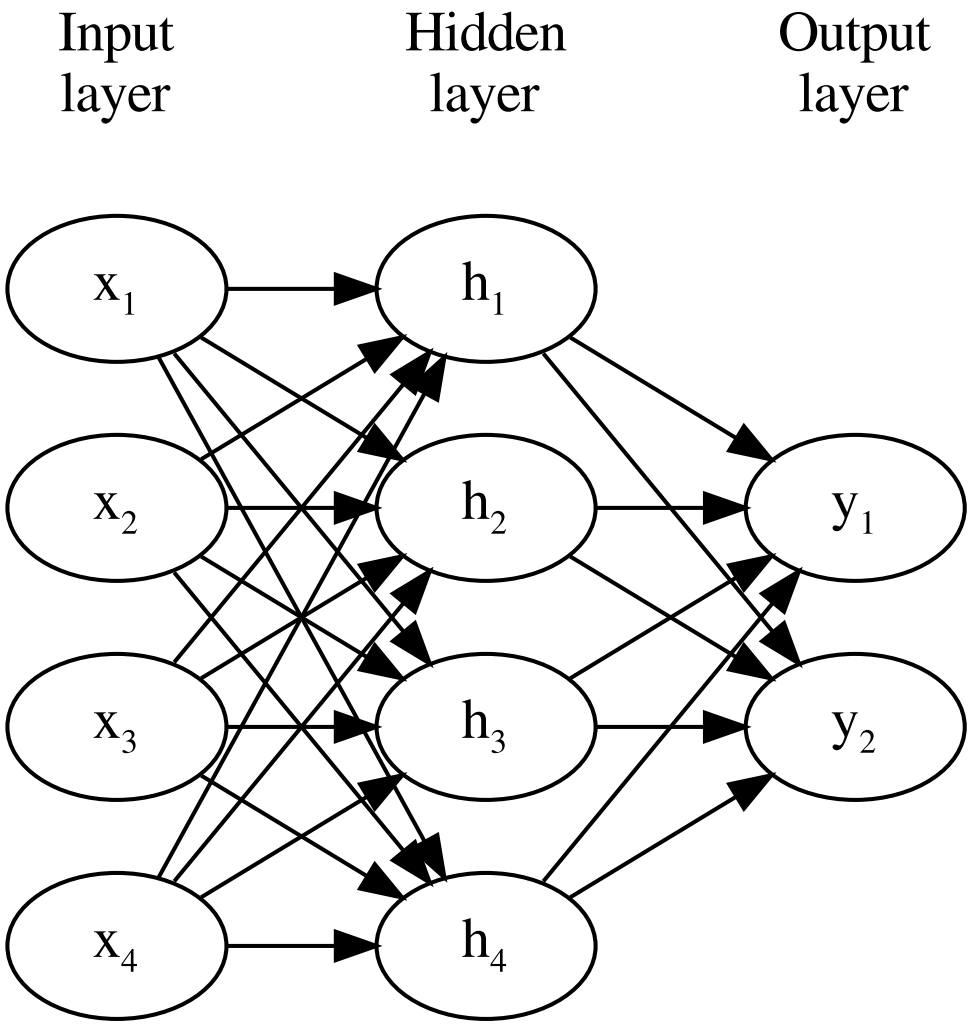
Let us have a dataset with training, validation, and test sets, each containing examples $(\boldsymbol{x}, y)$. Depending on y, consider one of the following output activation functions:

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

If $\boldsymbol{x} \in \mathbb{R}^D$, we can use a neural network with an input layer of size D, 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* $\mathrm{softmax}$*; 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 – Single-Hidden-Layer MLP



We have

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

where

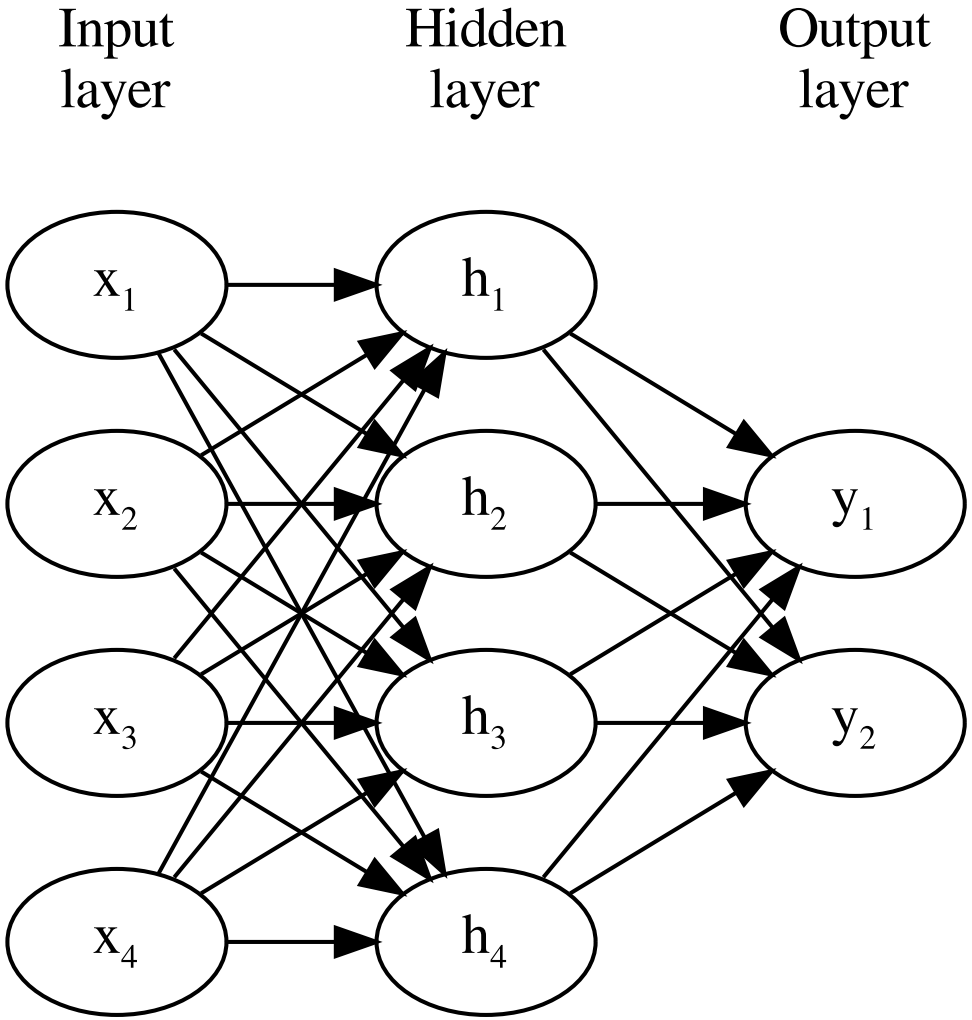
- $\boldsymbol{W}^{(1)} \in \mathbb{R}^{D \times H}$ is a matrix of **weights**,
- $\boldsymbol{b}^{(1)} \in \mathbb{R}^{H}$ is a vector of **biases**,
- $f^{(1)}$ is an activation function.

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

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

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 a step of the SGD algorithm (or any of its adaptive variants), updating the parameters to minimize the loss $E(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\hat{p}_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$ derived by MLE:

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

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

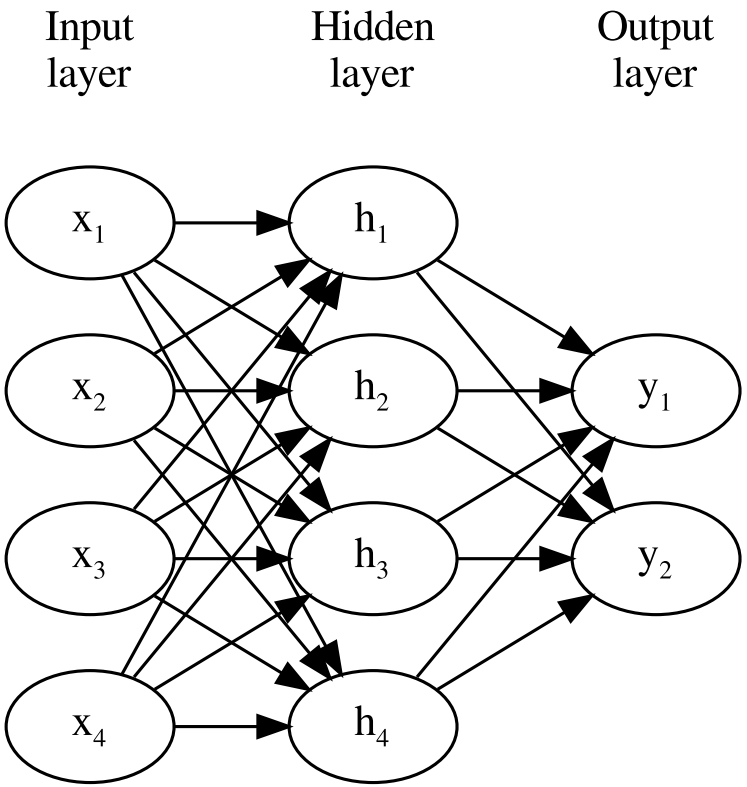
$$\boldsymbol{H} = f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right),$$
$$\boldsymbol{O} = f^{(2)}\left(\boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right) = f^{(2)}\left(f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)\boldsymbol{W}^{(2)}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

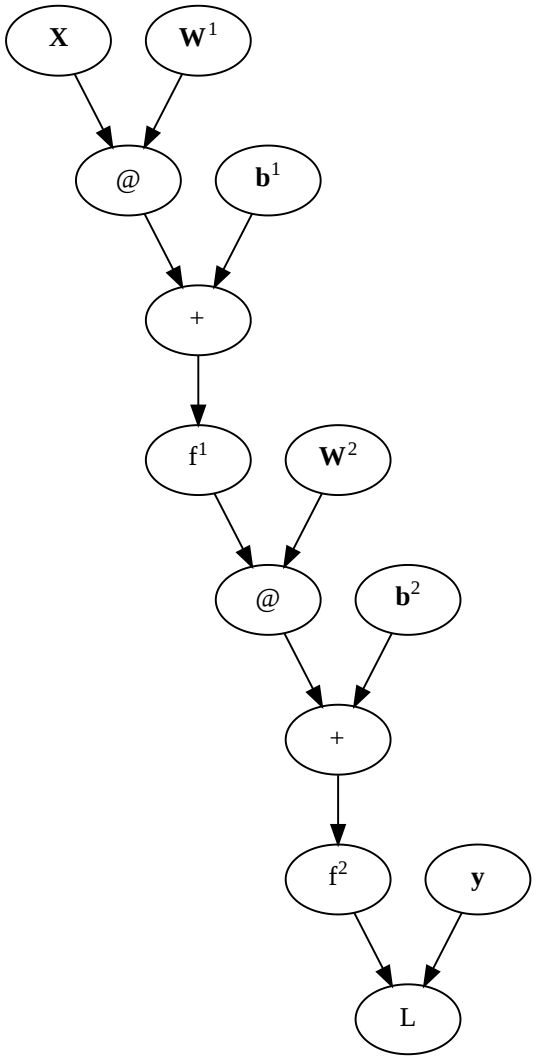
  The derivatives

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

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

Putting It All Together – Computation Graph





Putting It All Together – Designing and Training NNs

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)	Deep Learning
Architecture	⋮⋮⋮	⋮⋮⋮⋮⋮⋮⋮⋮⋮ CNN, RNN, Transformer, VAE, GAN, ...
Activation func.	$\tanh, \sigma$	$\tanh$, ReLU, LReLU, GELU, Swish (SiLU), SwiGLU, ...
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, ...

Regularization

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.

We now describe several basic regularization methods:

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

Regularization – Early Stopping

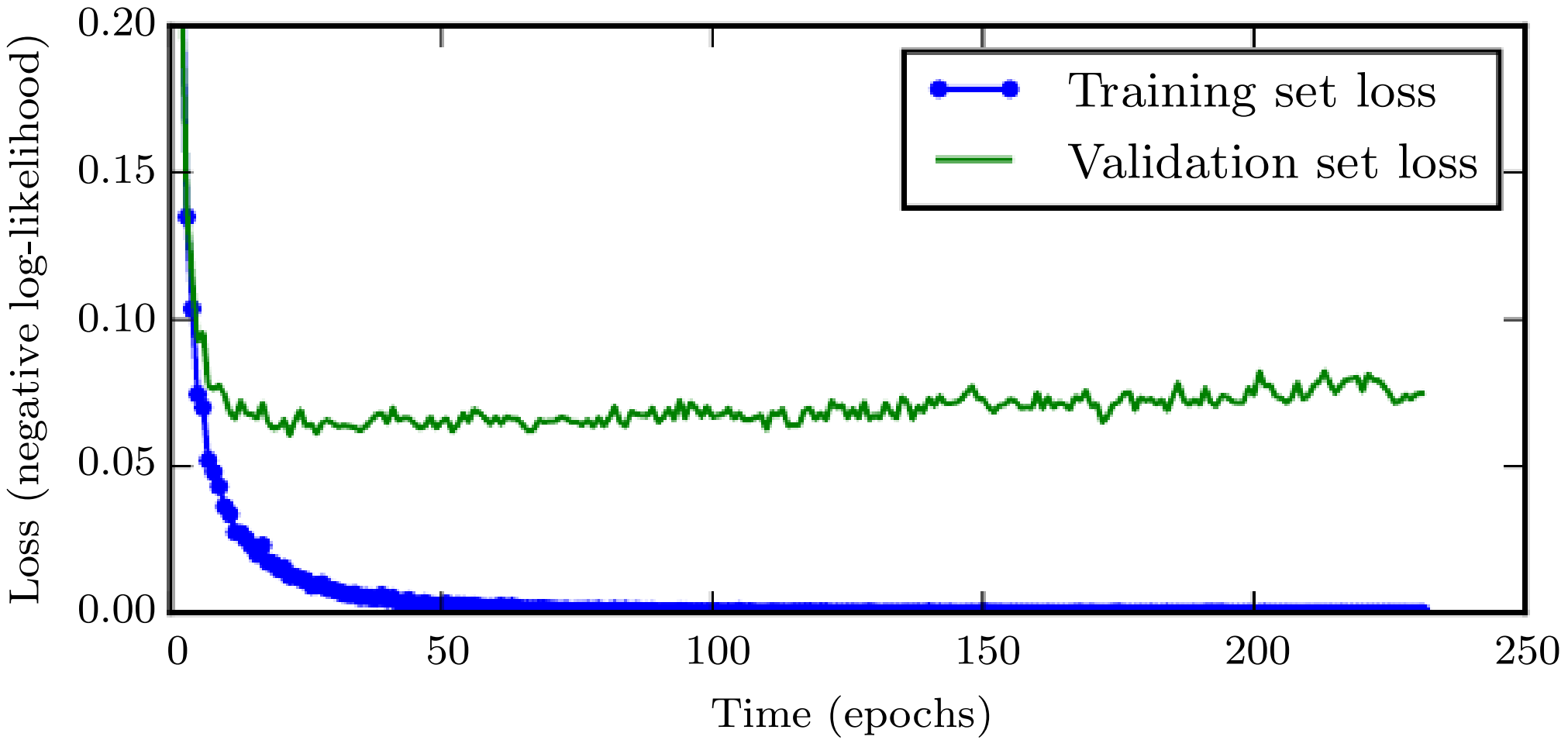


Figure 7.3 of "Deep Learning" book, https://www.deeplearningbook.org

L2 Regularization

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

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

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

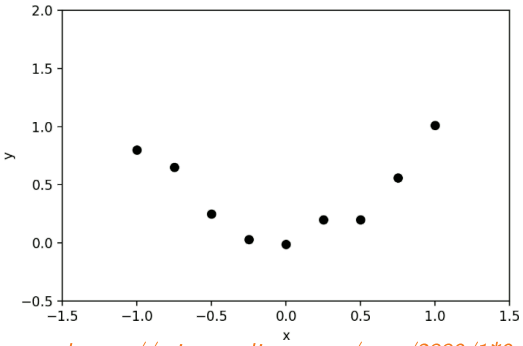
for a suitable (usually very small) λ.

Note that the L^2-regularization is usually not applied to the *bias*, only to the "proper" weights, because bias parameters usually do not influence the sharpness of the predictions.

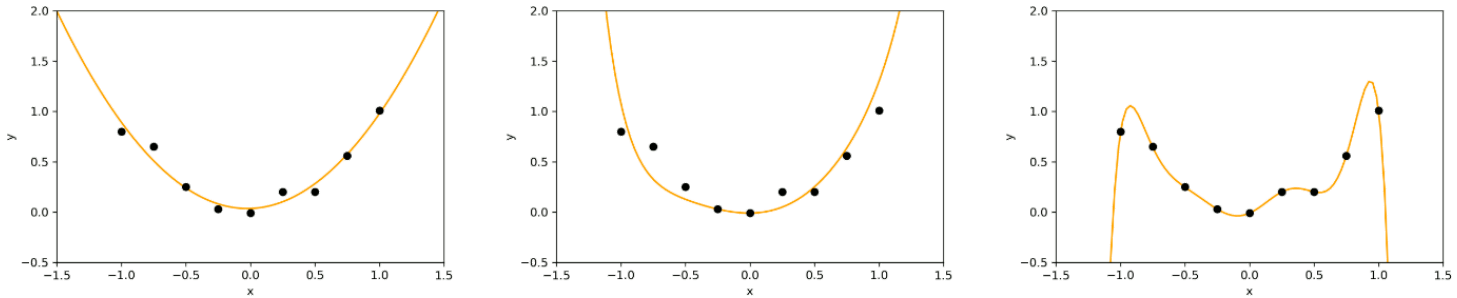
L2 Regularization

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 $\frac{\partial \boldsymbol{x}^T \boldsymbol{W} + \boldsymbol{b}}{\partial \boldsymbol{x}} = \boldsymbol{W}$).

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



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



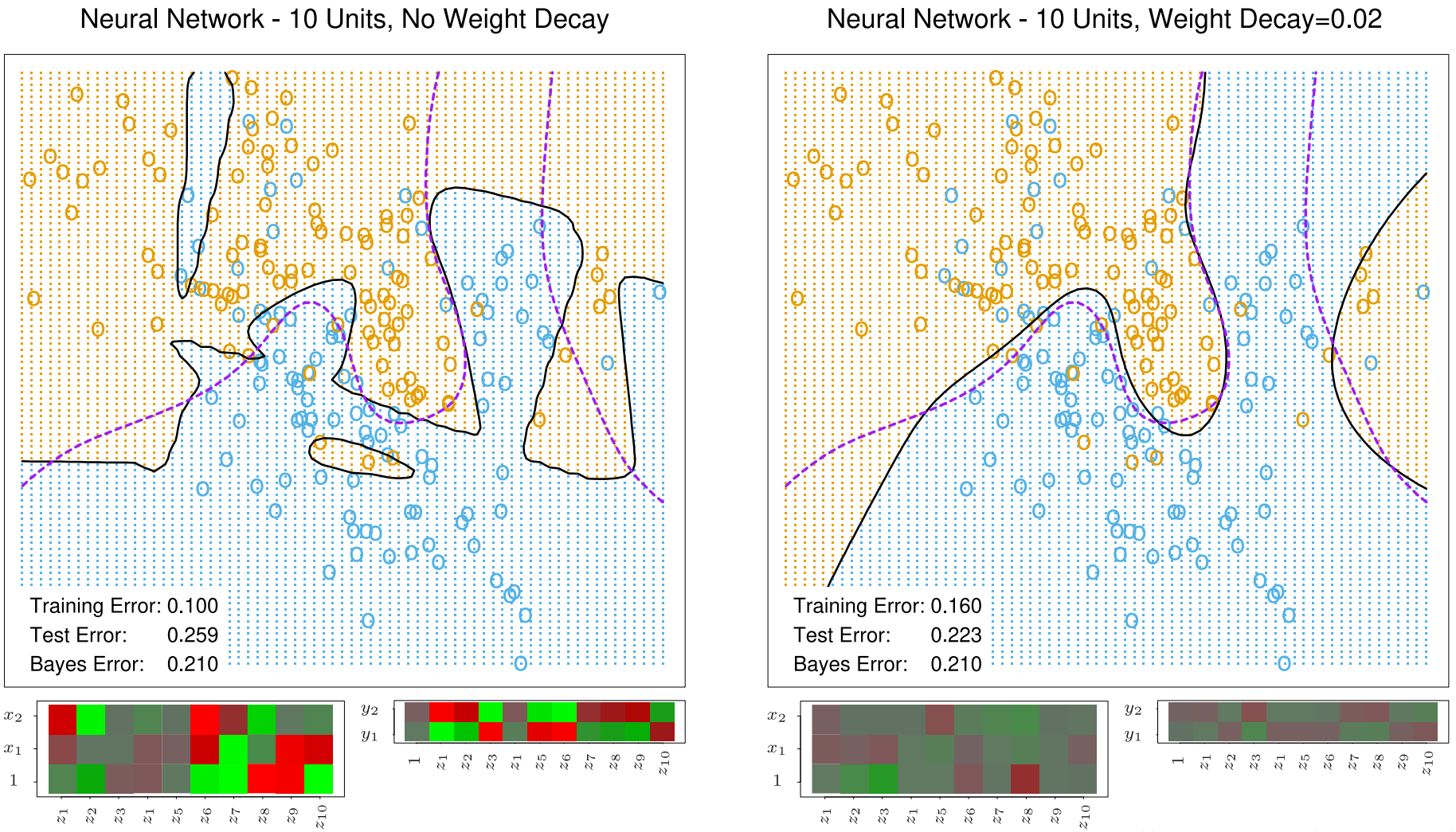
(a) #params = 3	(b) #params = 9	(c) #params = 9
MSE = 0.006	MSE = 0.035	MSE = 0
L2 norm = 0.90	L2 norm = 1.06	L2 norm = 32.69
L1 norm = 0.98	L1 norm = 2.32	L1 norm = 70.03

https://miro.medium.com/max/2880/1*DVFYChNDMNlS_7CVq2PhSQ.png

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*UolRlKXikCz7SFsPfSZrYQ.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

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

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

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

L2 Regularization

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

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

This update can be rewritten to

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

Terminologically, 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

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

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 `torch.optim.AdamW` and `keras.optimizers.AdamW` and 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

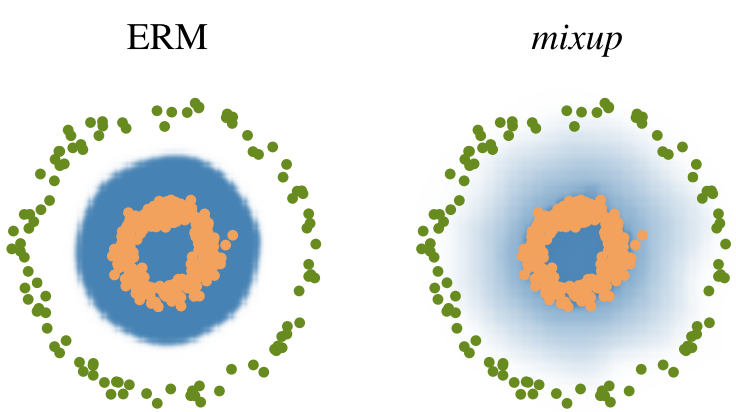
$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial EJ}{\partial \theta_i} - \min\left(\alpha\lambda, |\theta_i|\right) \operatorname{sign}(\theta_i).$$

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, ...
  - AutoAugment, RandAugment
  - Mixup (appeared in 2017), CutMix (published in 2019)



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

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

	ResNet-50	Mixup [48]	Cutout [3]	CutMix
Image				
Label	Dog 1.0	Dog 0.5 Cat 0.5	Dog 1.0	Dog 0.6 Cat 0.4
ImageNet Cls (%)	76.3 (+0.0)	77.4 (+1.1)	77.1 (+0.8)	**78.6** **(+2.3)**
ImageNet Loc (%)	46.3 (+0.0)	45.8 (-0.5)	46.7 (+0.4)	**47.3** **(+1.0)**
Pascal VOC Det (mAP)	75.6 (+0.0)	73.9 (-1.7)	75.1 (-0.5)	**76.7** **(+1.1)**

Figure 1 of "CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features", https://arxiv.org/abs/1905.04899

- 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 is that if models have uncorrelated (independent) errors, then by averaging model outputs, the errors 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 $\mathbb{E}\left[(y_i(\boldsymbol{x}) - y)^2\right] = \mathbb{E}\left[\varepsilon_i^2(\boldsymbol{x})\right]$.

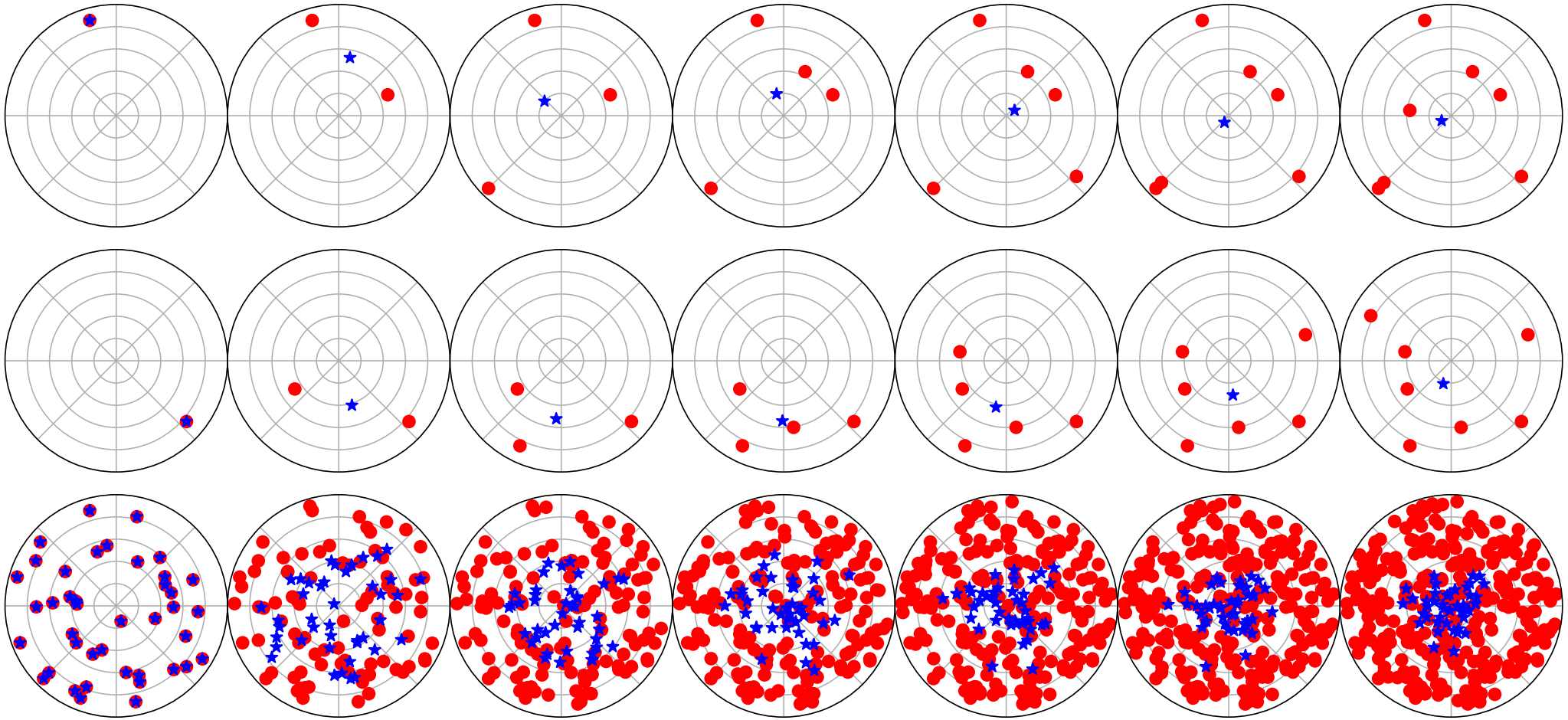
Because for uncorrelated identically distributed random variables x_i we have

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

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

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.

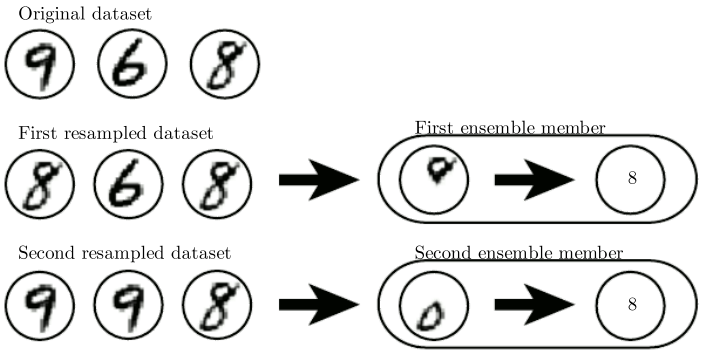


Figure 7.5 of "Deep Learning" book, https://www.deeplearningbook.org

However, ensembling usually has high performance requirements.

Dropout

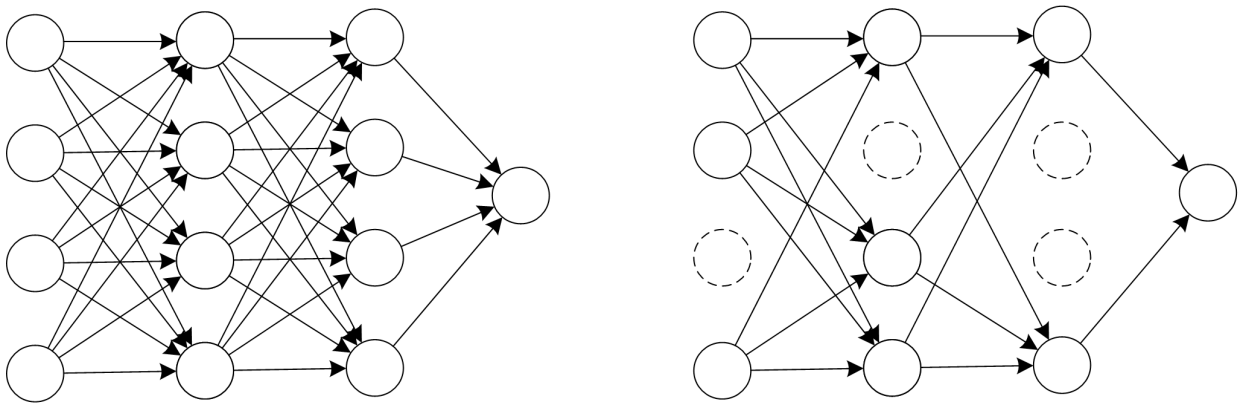
Regularization – Dropout

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.



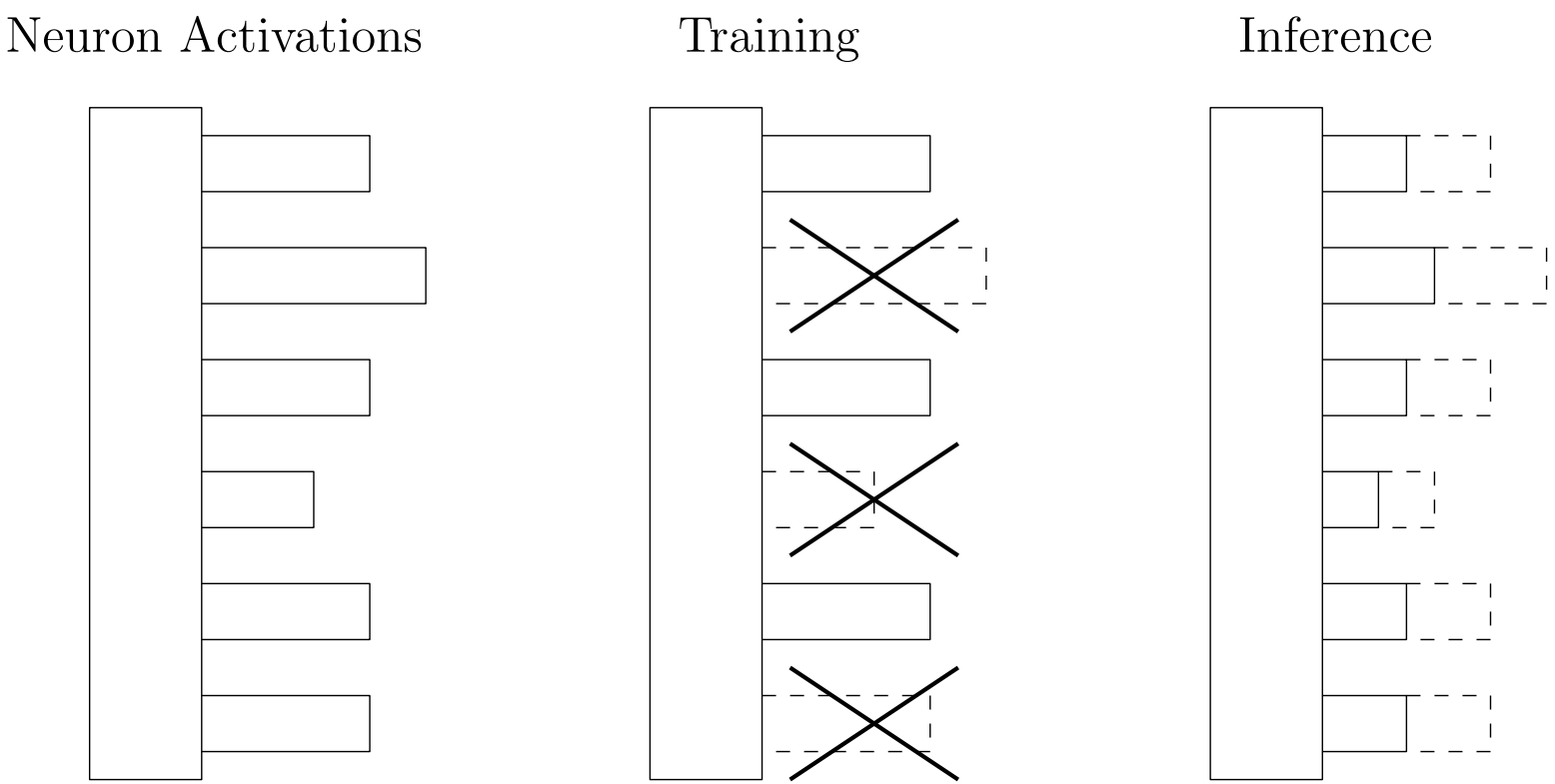
(a) Standard Neural Network

(b) Network after Dropout

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

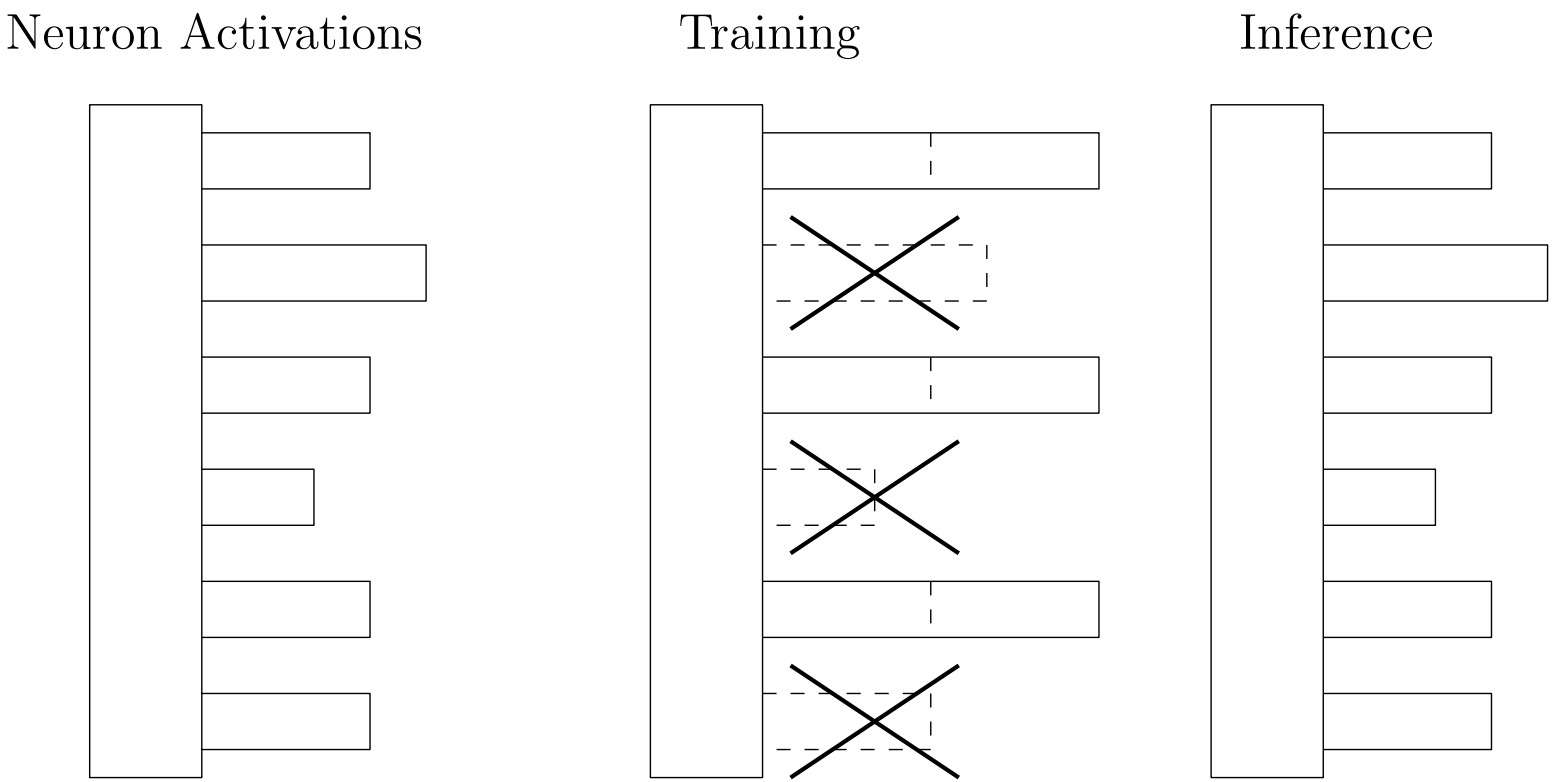
Regularization – Dropout

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

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

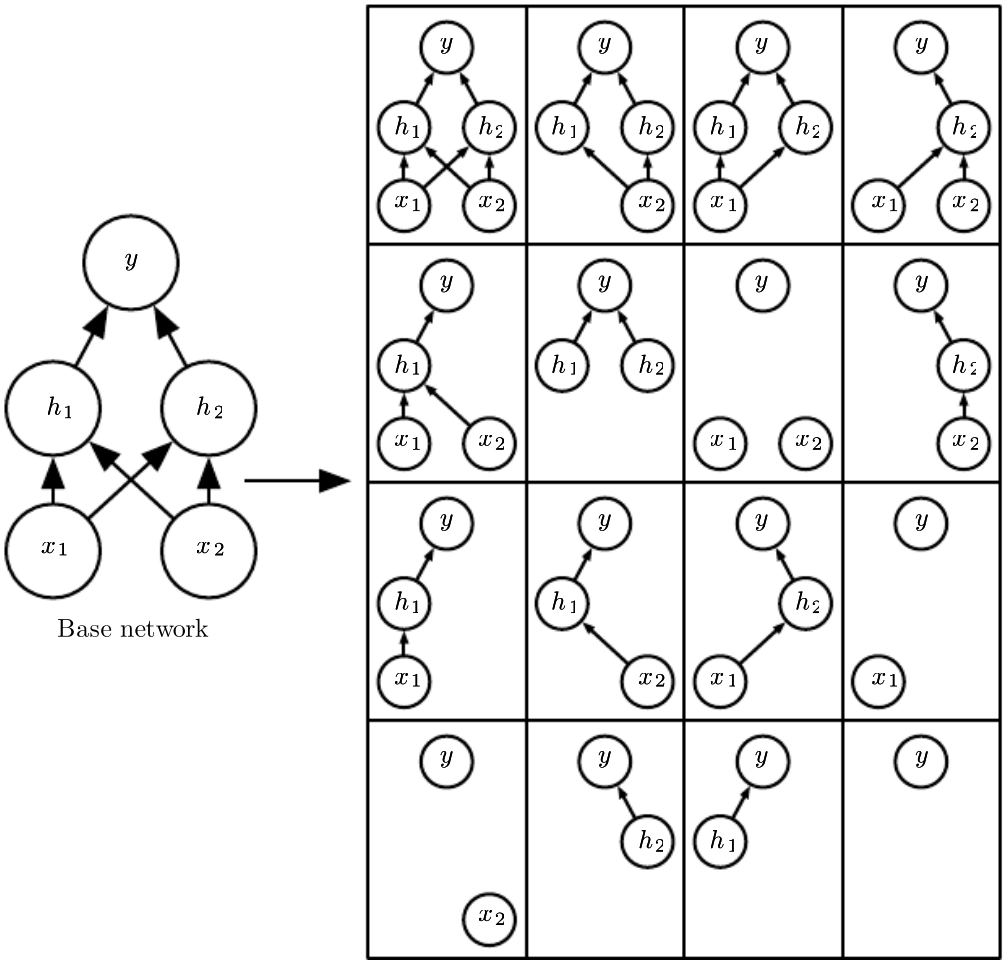


Figure 7.6 of "Deep Learning" book, https://www.deeplearningbook.org

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:

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

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

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

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

Regularization – Dropout Implementation

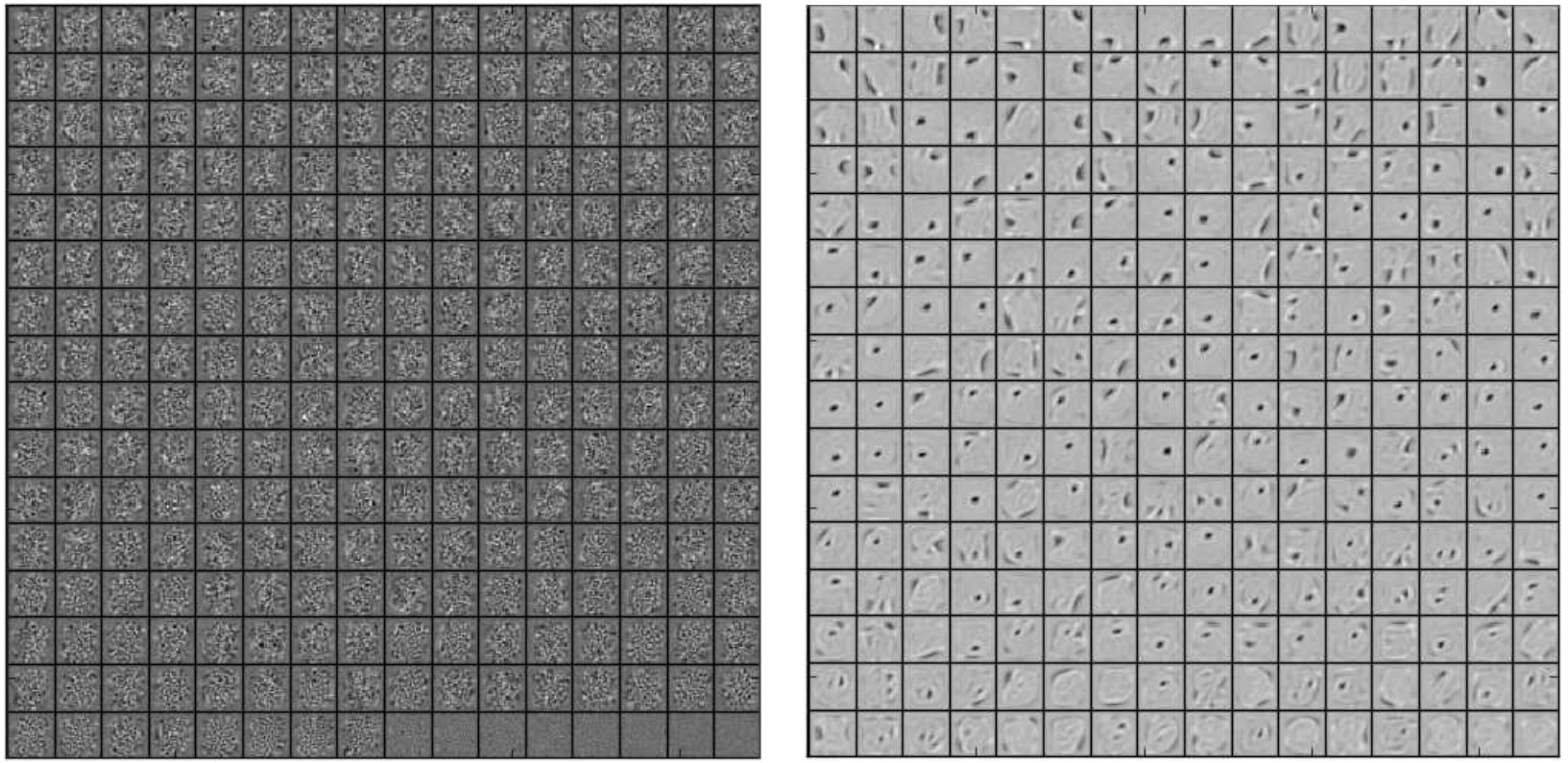
The following is a simplified example implementation of `torch.nn.functional.dropout`:

```
def dropout(inputs, p=0.5, training=True):
    def do_inference():
        return inputs

    def do_train():
        random_noise = torch.rand(inputs.shape)
        mask = (random_noise >= p).to(inputs.dtype)
        return inputs * mask / (1 - p)

    if training and p != 0.0:
        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

Label Smoothing

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.

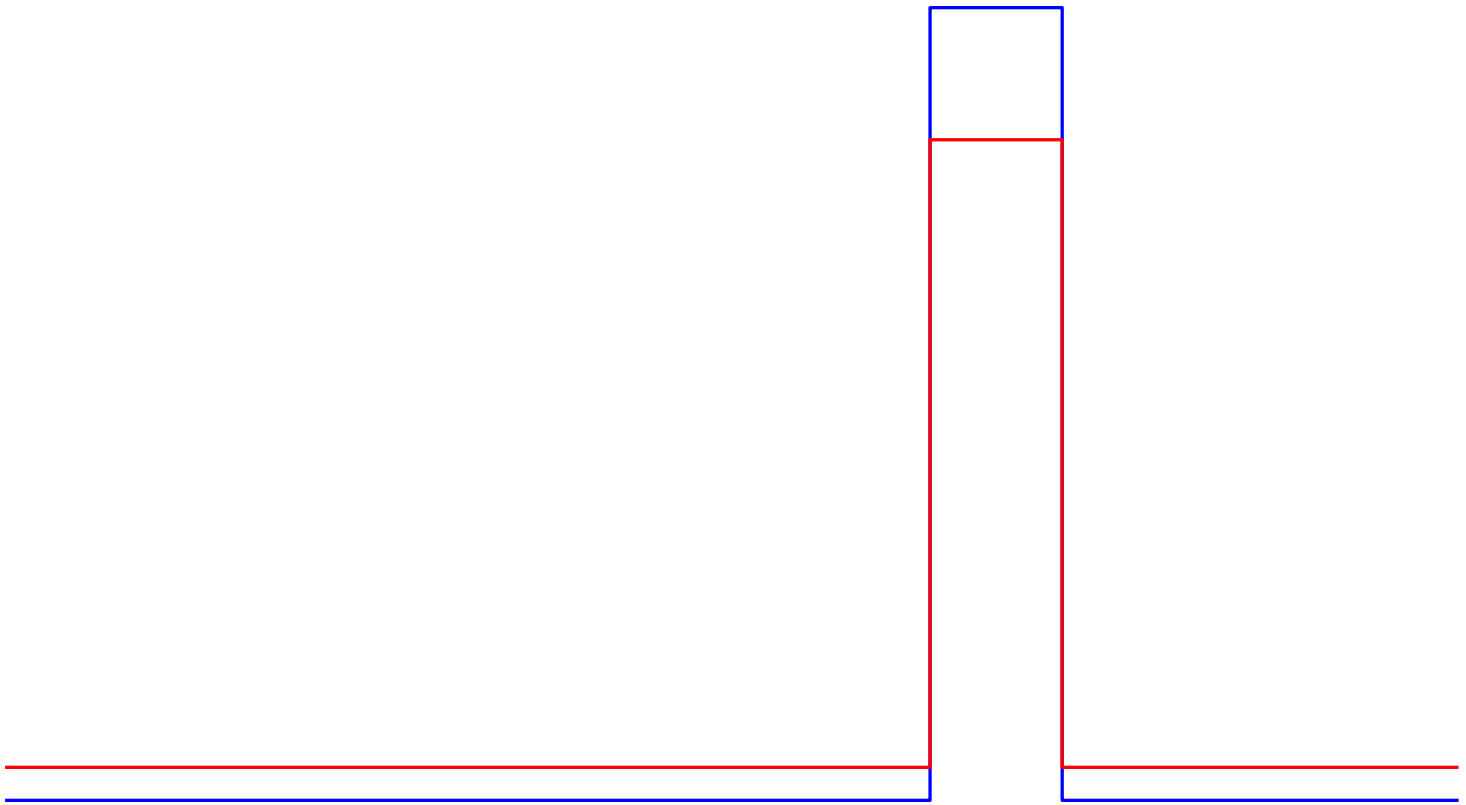
Ideally, we would like a full (non-sparse) categorical distribution of classes for training examples, but that is usually not available.

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

In the case of classification with the gold class *gold*, the target categorical 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 starting dropout rate is 0.5 if your model has enough capacity,
  - otherwise, use 0.3-0.1 if the model is underfitting;
- 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 of Neural Network Training

Convergence

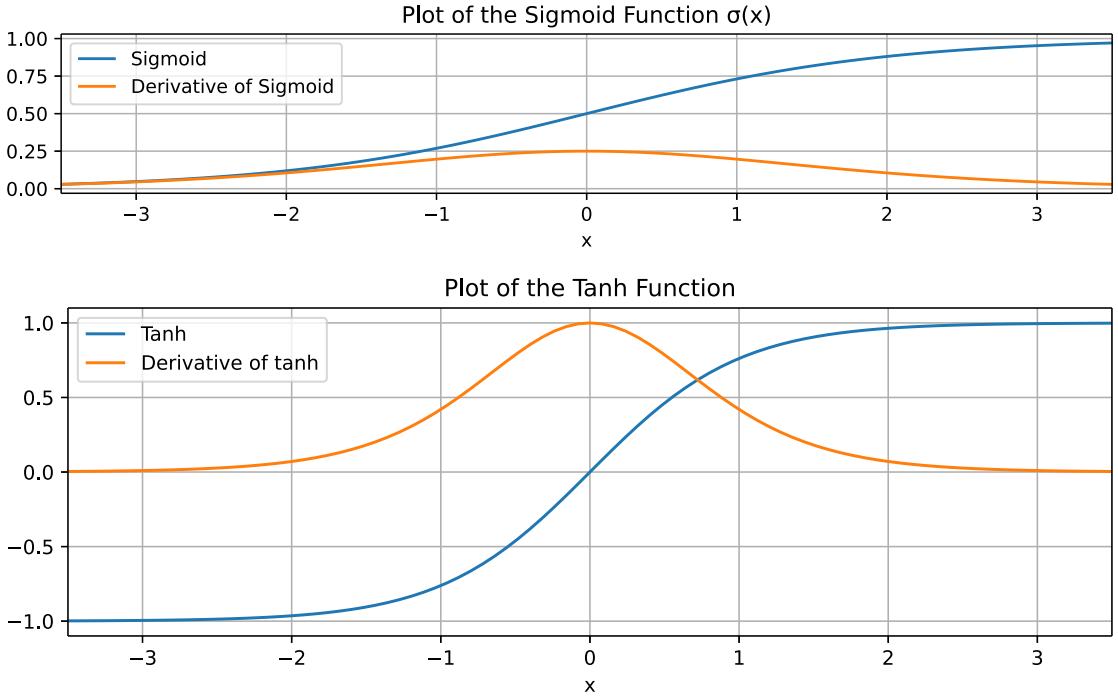
The training process might or might not converge. Even if it does, it might converge slowly or quickly.

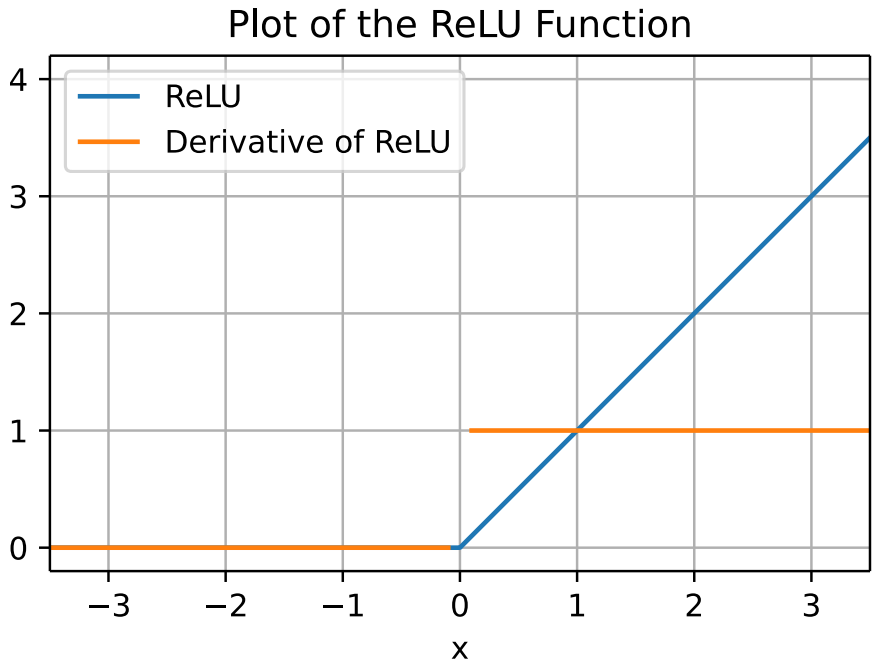
A major issue of convergence of deep networks is to make sure that the gradient with respect to all parameters is reasonable at all times, i.e., it does not decrease or increase too much with depth or in different batches.

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





Hidden Layer Interpretation

Considering a network with a single hidden layer:

- The last part (from the hidden layer to the output layer) is a linear model, which can distinguish linearly separable data only.
- The part from the inputs to the hidden layer can be considered as automatically constructed features. The features are a linear mapping of the input values followed by a nonlinearity, and the Universal approximation theorem proves they can always be constructed to achieve as good a fit of the training data as is required.

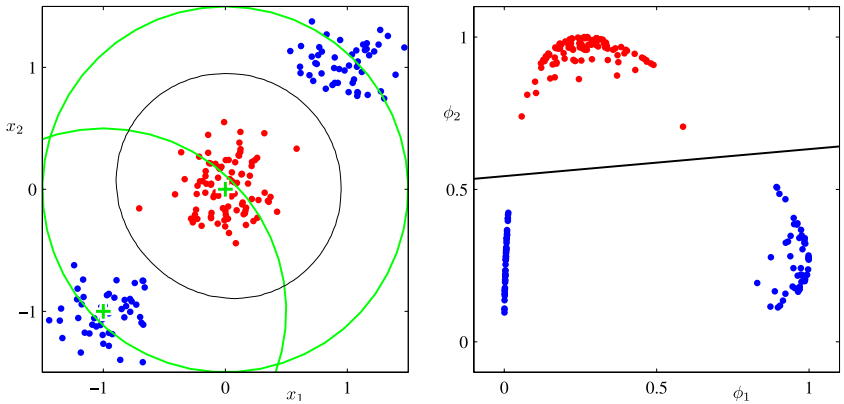


Figure 4.12 of Pattern Recognition and Machine Learning.

However, the weights in the first layer of such a MLP must be initialized randomly. If we used just zeros, all the constructed features (hidden layer nodes) would behave identically and we would never distinguish them.

Using random weights corresponds to starting with random features, which allows the SGD to make progress (improve the individual features).

Convergence – Parameter Initialization

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

- Biases should be initialized to 0 (Keras, TF, Jax; for some reason not PyTorch).
- 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.
    - 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

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

Convergence – Parameter Initialization

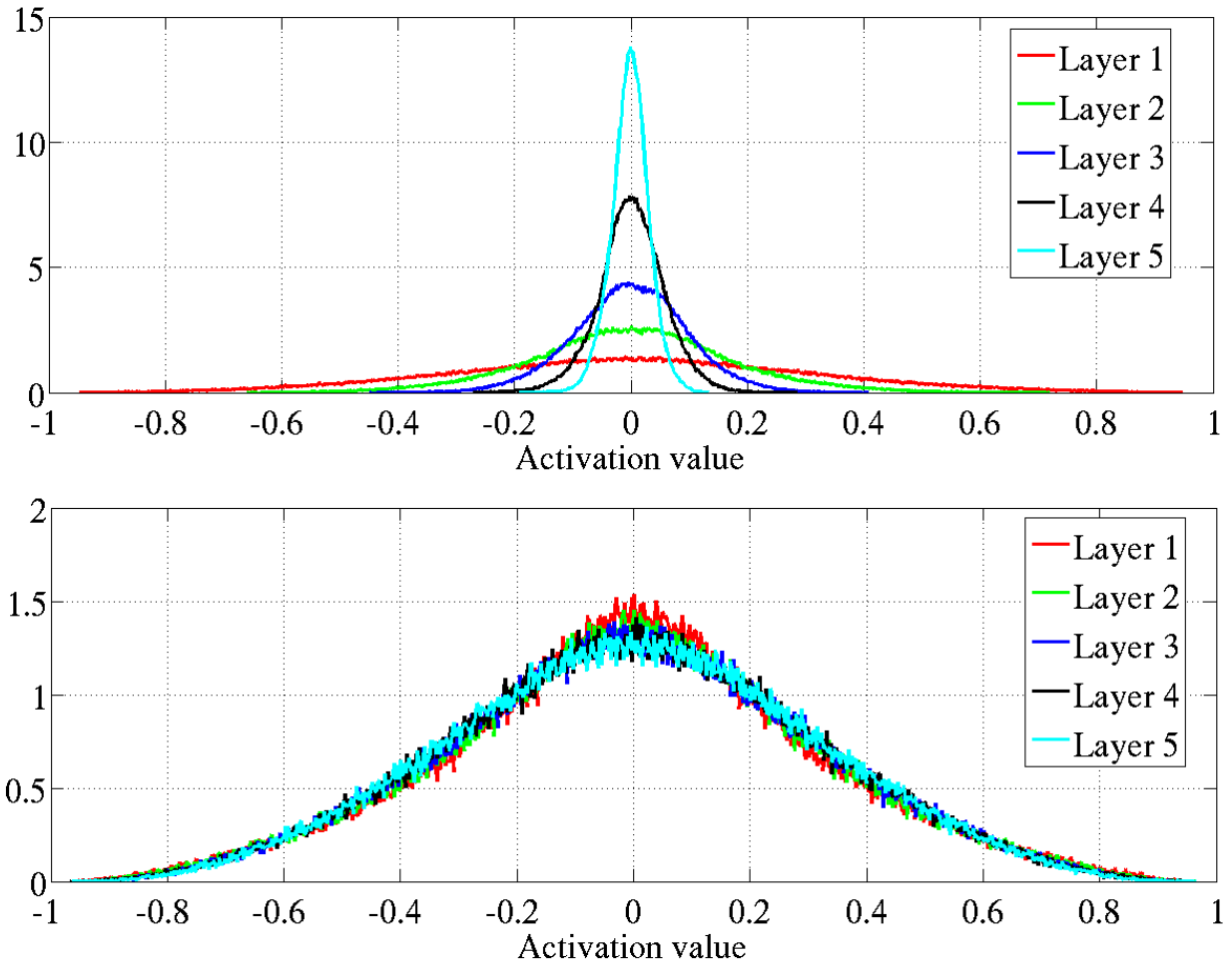


Figure 6 of "Understanding the difficulty of training deep feedforward neural networks", http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Convergence – Parameter Initialization

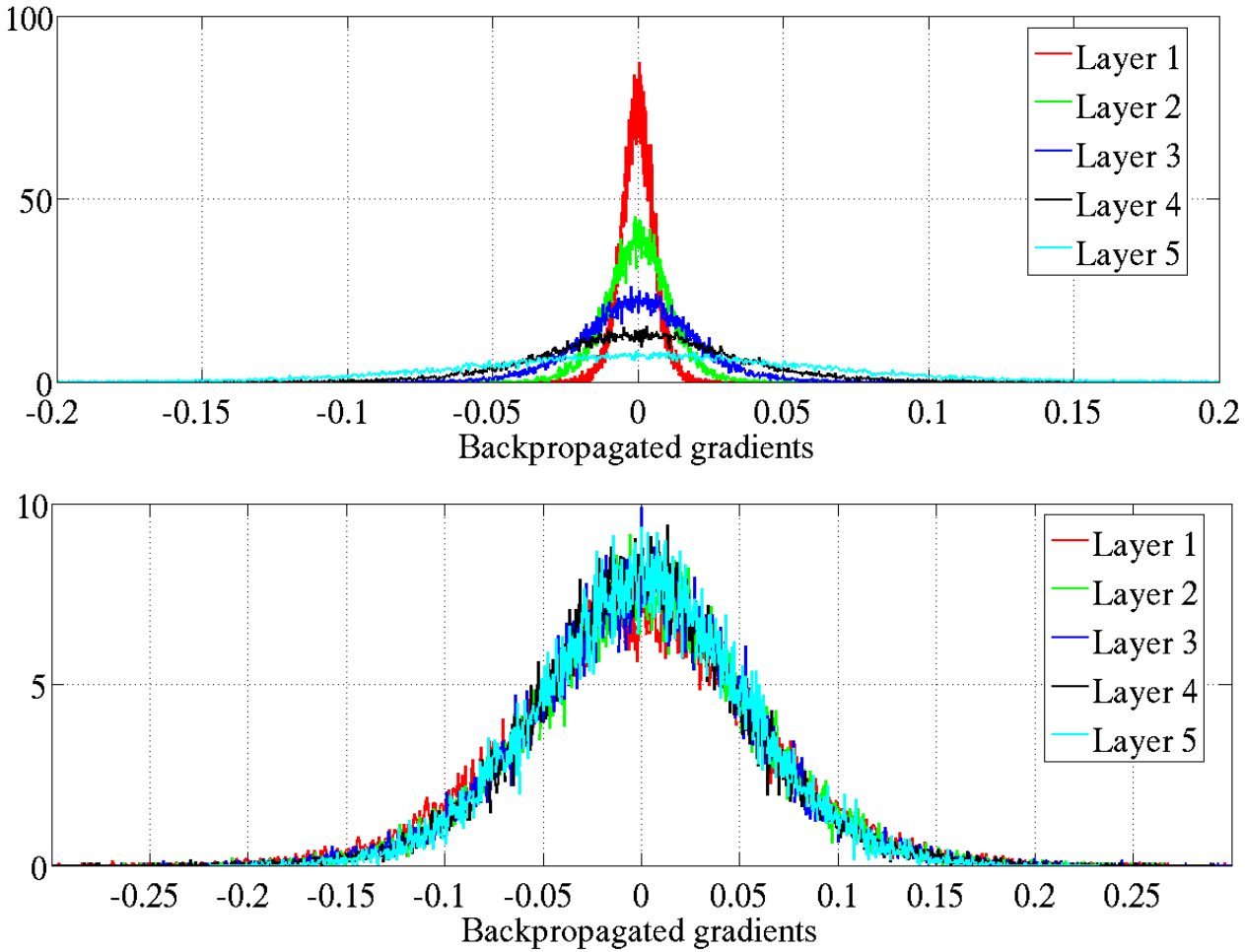


Figure 7 of "Understanding the difficulty of training deep feedforward neural networks", http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Convergence – Gradient Clipping

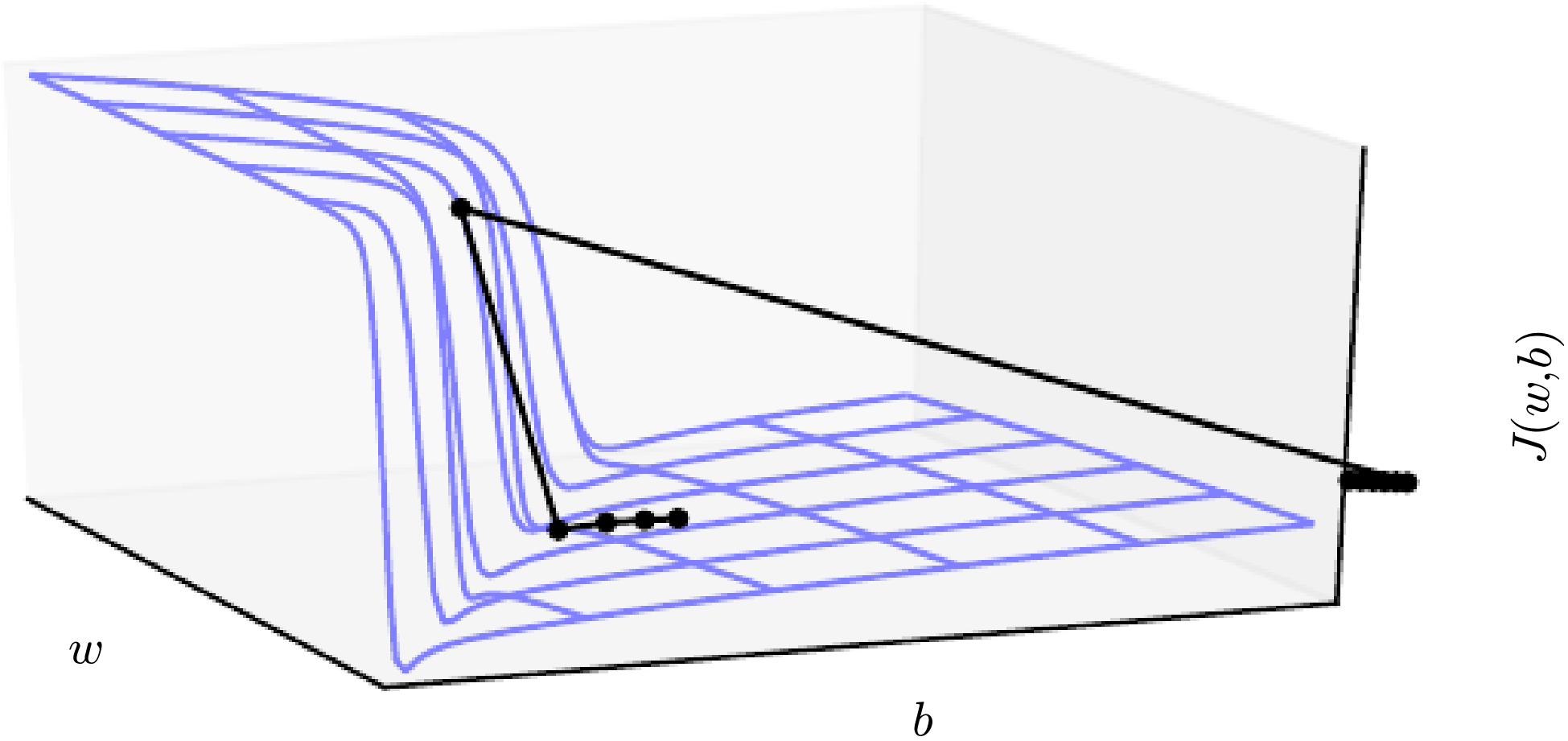


Figure 8.3 of "Deep Learning" book, https://www.deeplearningbook.org

Convergence – Gradient Clipping

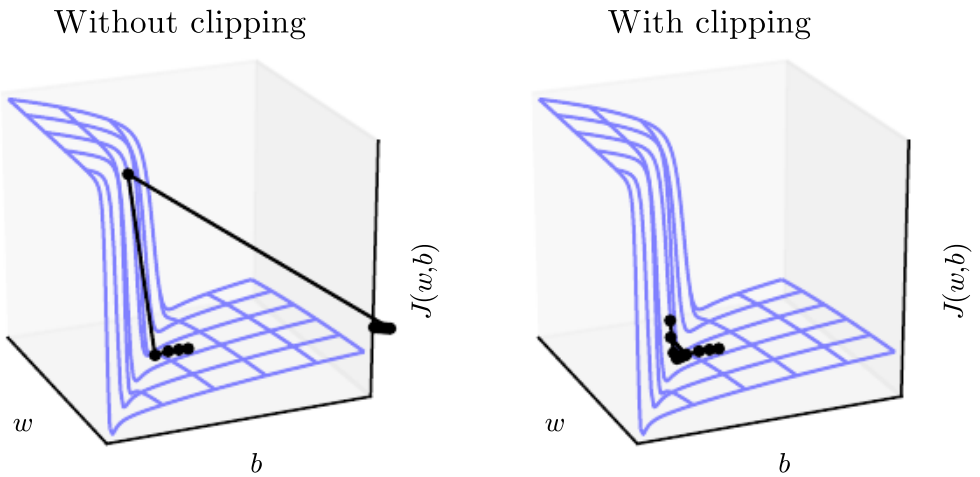


Figure 10.17 of "Deep Learning" book, https://www.deeplearningbook.org

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 single weight (`torch.nn.utils.clip_grad_value_`) or for the gradient as a whole (`torch.nn.utils.clip_grad_norm_`).

Derivative of the MLE Losses

Derivative of the MSE Loss

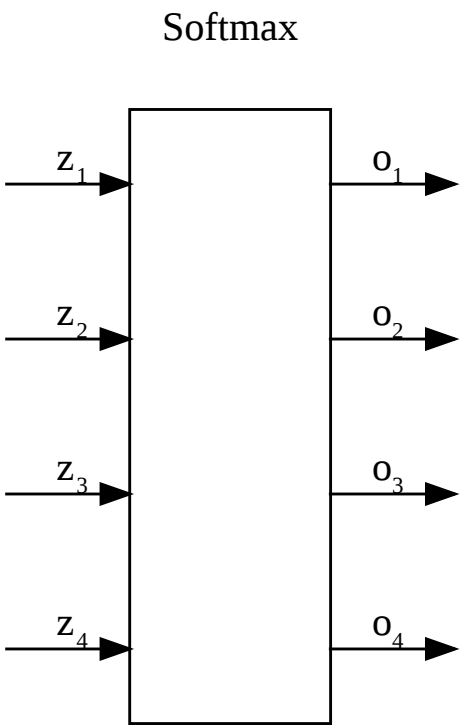
Given the MSE loss of

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

the derivative with respect to the model output is simply:

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

Derivative of the Softmax MLE Loss



Let us have a softmax output layer with

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

Derivative of the Softmax MLE Loss

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

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

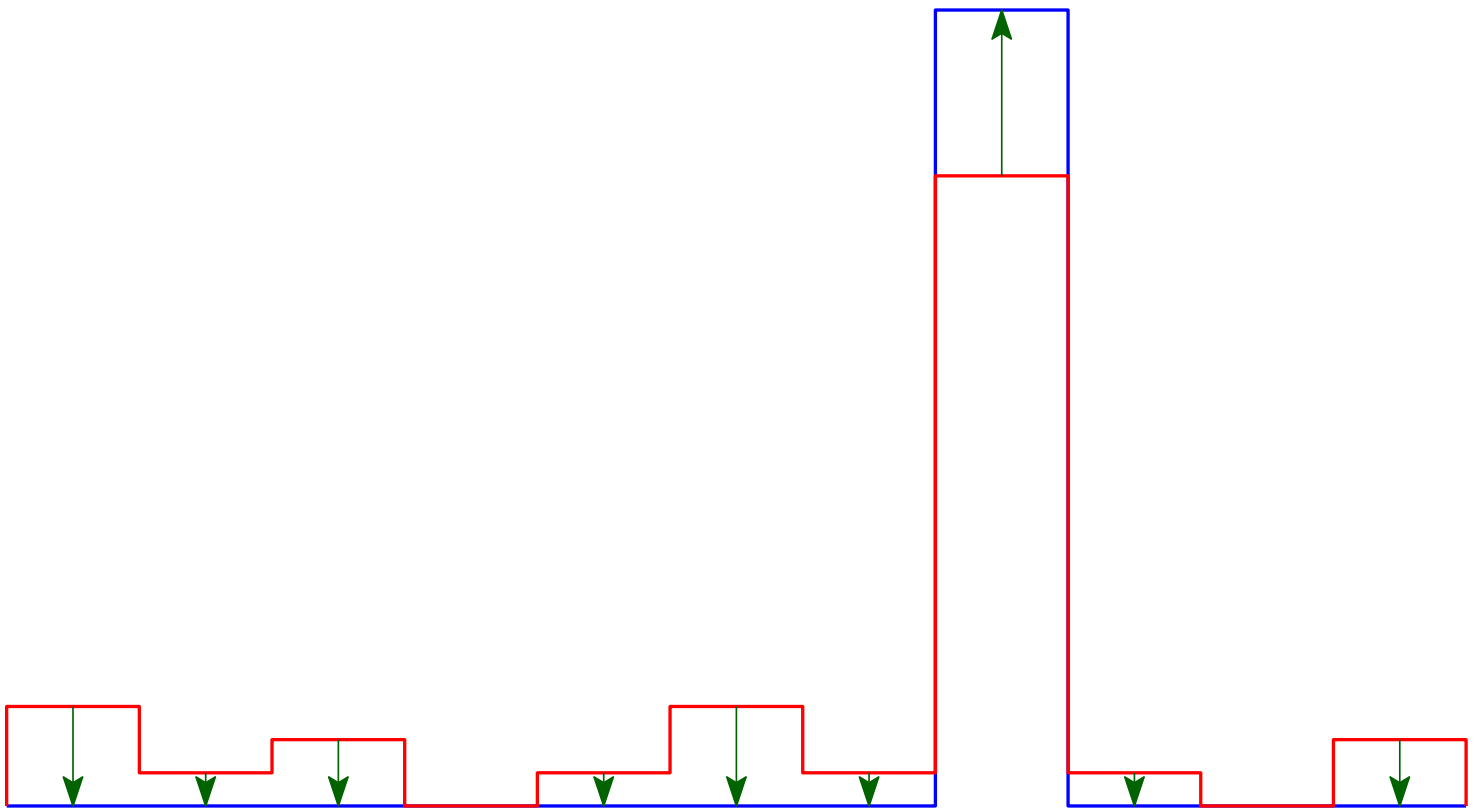
This is the **negative log likelihood** or **(sparse) categorical cross-entropy** loss.

The derivative 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 the one-hot encoding (a vector with 1 at the index $gold$ and 0 everywhere else).

Derivative of the Softmax MLE Loss



Gold distribution

Model distribution

Loss derivative with respect to the softmax inputs.

Derivative of the Softmax MLE Loss

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

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

This is the **(full) categorical cross-entropy** loss.

Reusing the result showing that $-\frac{\partial \log o_i}{\partial \boldsymbol{z}} = \boldsymbol{o} - \mathbf{1}_i$, we obtain

$$\frac{\partial L}{\partial \boldsymbol{z}} = -\sum_i g_i \frac{\partial \log o_i}{\partial \boldsymbol{z}} = \sum_i \left(g_i \cdot \boldsymbol{o} - g_i \cdot \mathbf{1}_i\right) = \boldsymbol{o} - \boldsymbol{g}.$$

Derivative of the Sigmoid MLE Losses

For binary classification, denoting $o \stackrel{\text{def}}{=} \sigma(z)$ and assuming gold label $g \in \{0, 1\}$, we have that

$$L\big(\sigma(z), g\big) = -\log p_{\text{model}}(g|o).$$

Recalling the Bernoulli distribution probability $p(x; \varphi) = \varphi^x (1-\varphi)^{1-x}$, we obtain the **binary cross-entropy** loss:

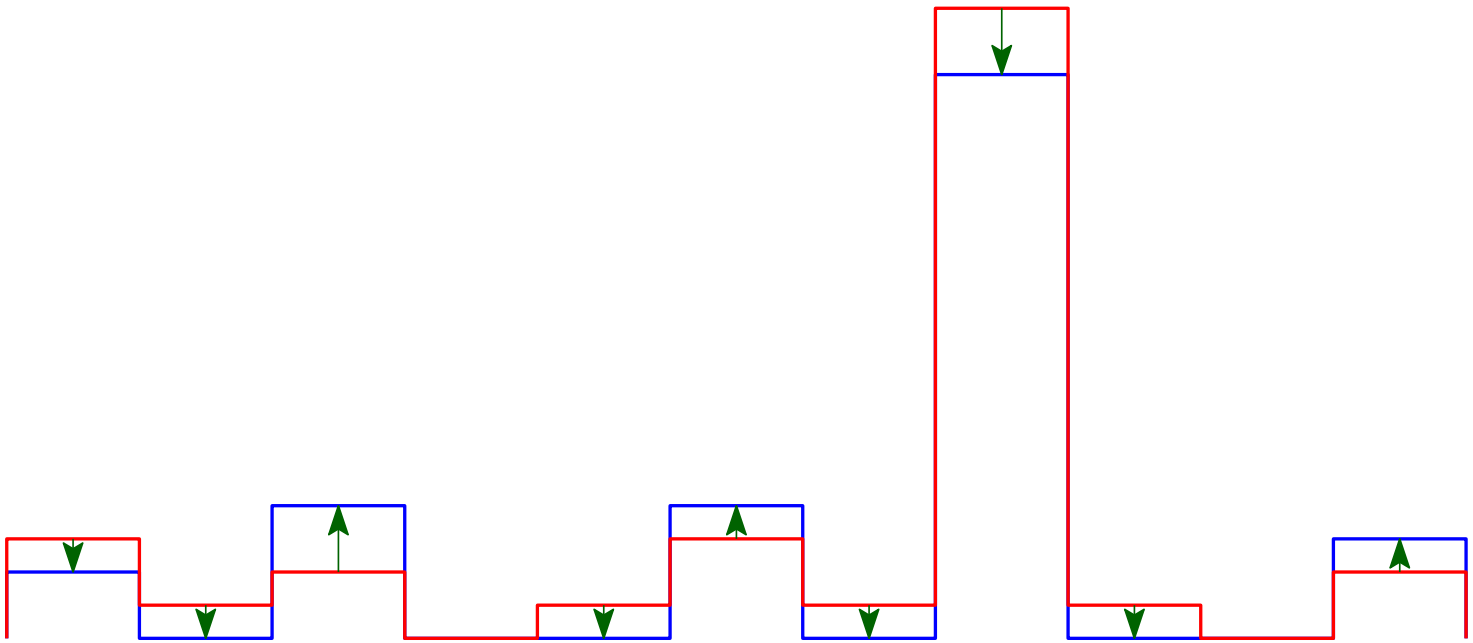
$$L\big(\sigma(z), g\big) = -\log \left(o^g (1-o)^{1-g}\right) = -g \log o - (1-g) \log(1-o).$$

Analogously to the $\mathrm{softmax}$ MLE derivatives, we get that $\frac{\partial L}{\partial z} = o - g$.

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

$$\mathrm{softmax}\left([0 \;\; x]\right)_1 = \frac{e^x}{e^x + e^0} = \frac{1}{1 + e^{-x}} = \sigma(x).$$

Derivative of Softmax MLE Loss



Gold distribution

Model distribution

Loss derivative with respect to the softmax inputs.

Metrics and Losses

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 PyTorch, the losses are available in the `torch.nn` and `torch.nn.functional` modules.

However, no built-in metrics are provided (in contrast to for example Keras). Therefore, we will use the metrics from the `torchmetrics` package.

PyTorch Losses

The PyTorch losses offer two interfaces: an object one through the subclasses of `torch.nn.Module`, and a functional one via methods in the `torch.nn.functional` module. (Most modules offer their functionality also as a stateless function.)

Considering the mean squared error, the loss object can be constructed using

```
torch.nn.MSELoss(reduction="mean")
```

and the instances then provide a method

```
__call__(y_pred: torch.Tensor, y_true: torch.Tensor) -> torch.Tensor
```

returning a *reduced* loss value.

The possible `reductions` are

- `reduction="mean"`, producing a single scalar tensor;
- `reduction="sum"`, producing again a single scalar tensor;
- `reduction="none"`, producing a tensor of the original shape.

PyTorch Cross-entropy Losses

The landscape of cross-entropy losses provided by PyTorch is not particularly well designed.

- The cross-entropy of a categorical distribution is computed by

```
class torch.nn.CrossEntropyLoss(torch.nn.Module):
  def __init__(ignore_index=-100, label_smoothing=0, reduction="mean")
```

The resulting instance provides a method

```
__call__(y_pred: torch.Tensor, y_true: torch.Tensor) -> torch.Tensor
```

where:

- `y_pred` are the prediction **logits** with shape $[C]$, $[N, C]$, or $[N, C, d_1, \ldots, d_k]$;
- `y_true` are either:
  - the gold class indices with shape $[]$, $[N]$, $[N, d_1, \ldots, d_k]$, or
  - the gold distribution with shape $[C]$, $[N, C]$, $[N, C, d_1, \ldots, d_k]$.
- when `y_true` are class indices, the ones equal to `ignore_index` are ignored.

PyTorch Cross-entropy Losses

- A special-case of the `torch.nn.CrossEntropyLoss` is the

```
class torch.nn.NLLLoss(torch.nn.Module):
  def __init__(ignore_index=-100, reduction="mean")
  def __call__(y_pred: torch.Tensor, y_true: torch.Tensor) -> Tensor
```

  Compared to `torch.nn.CrossEntropyLoss`:
  - the `y_pred` must be **log-probability** (not logits), computable using for example `torch.nn.LogSoftmax` or `torch.nn.functional.log_softmax`;
  - the `y_true` always contains gold class indices;
  - label smoothing is not supported.

PyTorch Cross-entropy Losses

- The cross-entropy of a Bernoulli distribution can be computed by

```
class torch.nn.BCELoss(torch.nn.Module):
  def __init__(reduction="mean")
  def __call__(y_pred: torch.Tensor, y_true: torch.Tensor) -> Tensor
```

  where:
  - the `y_pred` must be **probabilities** (not logits neither log-probabilities),
  - the `y_true` are the gold probabilities.

  For numerical stability, the logarithms are clamped to -100 for very small/zero inputs.

- Alternatively, one might use

```
class torch.nn.BCEWithLogitsLoss(torch.nn.Module):
  def __init__(reduction="mean")
  def __call__(y_pred: torch.Tensor, y_true: torch.Tensor) -> Tensor
```

  where the `y_pred` must be **logits** instead of probabilities.

PyTorch Cross-entropy Losses

Apart from the object interface, functions computing the above losses are also provided:

- ```
  torch.nn.functional.mse_loss(y_pred, y_true, reduction="mean")
  ```
- ```
  torch.nn.functional.cross_entropy(y_pred, y_true,
    ignore_index=-100, label_smoothing=0, reduction="mean")
  ```
- ```
  torch.nn.functional.nll_loss(y_pred, y_true,
    ignore_index=-100, reduction="mean")
  ```
- ```
  torch.nn.functional.binary_cross_entropy(y_pred, y_true,
    reduction="mean")
  ```
- ```
  torch.nn.functional.binary_cross_entropy_with_logits(y_pred, y_true,
    reduction="mean")
  ```
  ```

Metrics

There are two important differences between metrics and losses.

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

The metrics in the `torchmetrics` package are subclasses of a `torchmetrics.Metric` class:

- ```
  class torchmetrics.Metric(torch.nn.Module):
    def update(y_pred : torch.Tensor, y_true: torch.Tensor) -> None
  ```
  updates the state of the metric by incorporating a batch of predictions and true outputs;
- ```
    def compute() -> torch.Tensor
  ```
  the `compute` method returns the current value of the metric;
- ```
    def reset() -> None
  ```
  the `reset` method clears the stored state of the metric.
- weirdly, the `forward(y_pred, y_true)` method (or calling the metric object directly) returns the metric of *only the passed batch*, but it also updates the stored metric state.
  ```

Common torchmetric Metrics

The `torchmetrics` package provides 100+ losses. The most common ones are:

- `torchmetrics.MeanMetric` computing averaged mean;
- `torchmetrics.MeanSquaredError` computing the mean squared error;
- `torchmetrics.Accuracy(task: Literal["binary","multiclass","multilabel"])` is a wrapper constructing a task-specific accuracy.

Common torchmetric Accuracy Metrics Variants

- `torchmetrics.Accuracy(task="binary", threshold=0.5, ...)`

  computes accuracy of binary classification from predicted **probabilities**;
  - if one of the inputs is not in $[0, 1]$ range, σ is applied to the batch 😱
  - originally I thought passing `threshold=0.0` would allow processing logits; however, the broken σ application means logits cannot be processed reliably by this metric;

- `torchmetrics.Accuracy(task="multiclass", num_classes, ignore_index=None, ...)`

  computes accuracy of classification into the given `num_classes`; the predictions can be either integral predicted classes or probabilities/logits that are passed through an argmax;

- `torchmetrics.Accuracy(task="multilabel", num_labels, threshold=0.5, ignore_index=None, ...)`

  computes a multilabel classification, where the model is capable of predicting any number of classes, each being predicted independently as a binary classification.