NPFL114, Lecture 03

Training Neural Networks II

Milan Straka

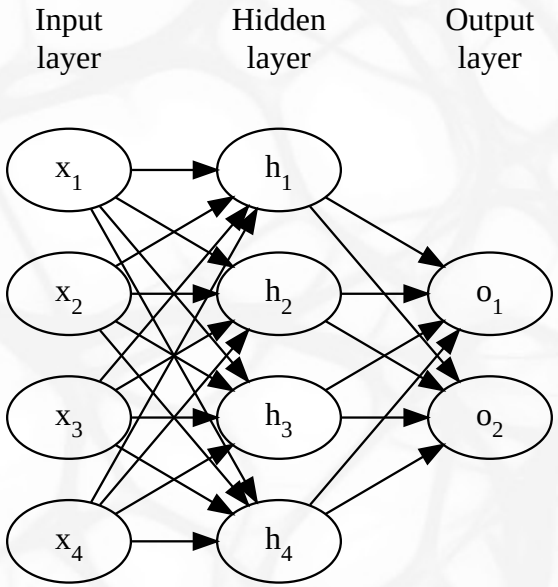
Putting It All Together

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

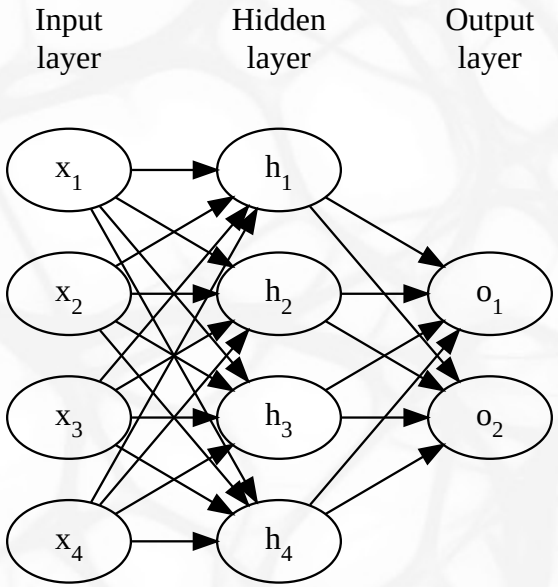


We have

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

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, and $f^{(1)}$ is an activation function.

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, and $f^{(2)}$ 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.

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.

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

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*.

Practical Issues

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

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

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

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

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

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

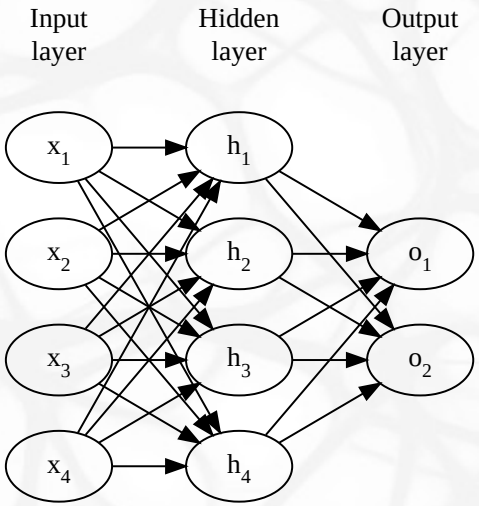
$$\boldsymbol{o} = f^{(2)}\left(\boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)}\right) = f^{(2)}\left(\boldsymbol{W}^{(2)}\left(f^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)\right) + \boldsymbol{b}^{(2)}\right)$$

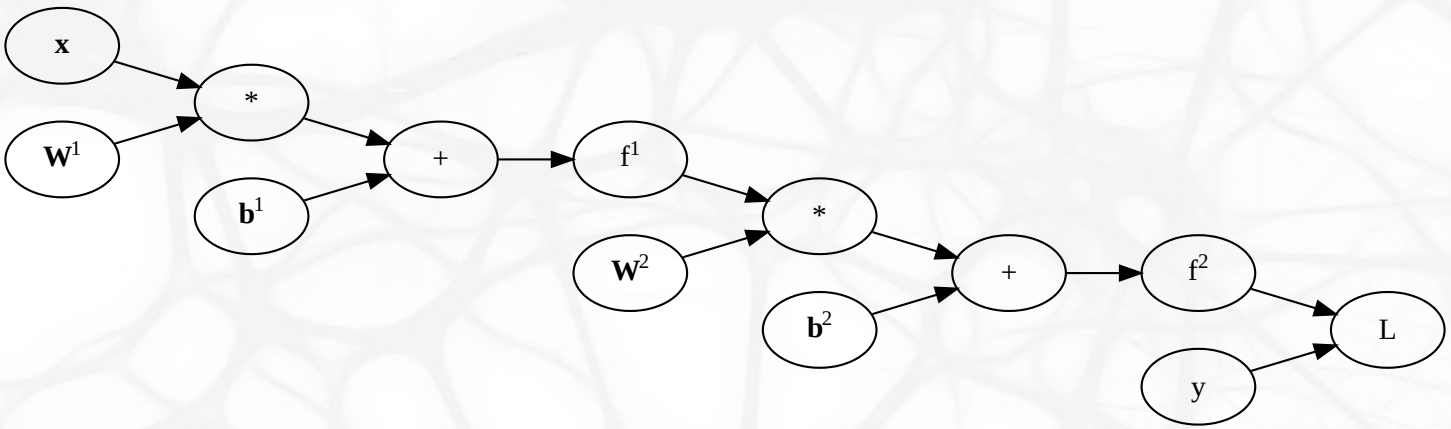
  The derivatives

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

  are then matrices (called *Jacobians*).

Computation Graph





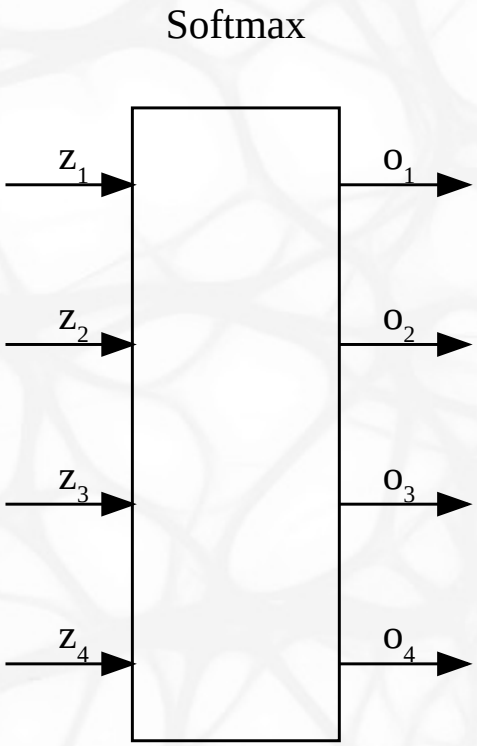
Neural Networks Web Browser Demos

- TensorFlow Playground
- Sketch RNN Demo
- DeepLearn.js Demos
  - DeepLearn.js Image Style Transfer Demo

High Level Overview

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

MLE Loss of Softmax



Let us have a softmax output layer with

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

MLE Loss of Softmax

Consider now the MLE estimation. The loss $L(\text{softmax}(\boldsymbol{z}), gold)$ 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, $\frac{\partial L}{\partial \boldsymbol{z}} = -\mathbf{1}_{gold} + \boldsymbol{o}$, where $\mathbf{1}_{gold}$ is 1 at index $gold$ and 0 otherwise.

MLE Loss of Softmax and Sigmoid

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

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

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

Adapting the previous procedure, we obtain

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

MLE Loss of Softmax and Sigmoid

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

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

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

Adapting the previous procedure, we obtain

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

Sigmoid

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

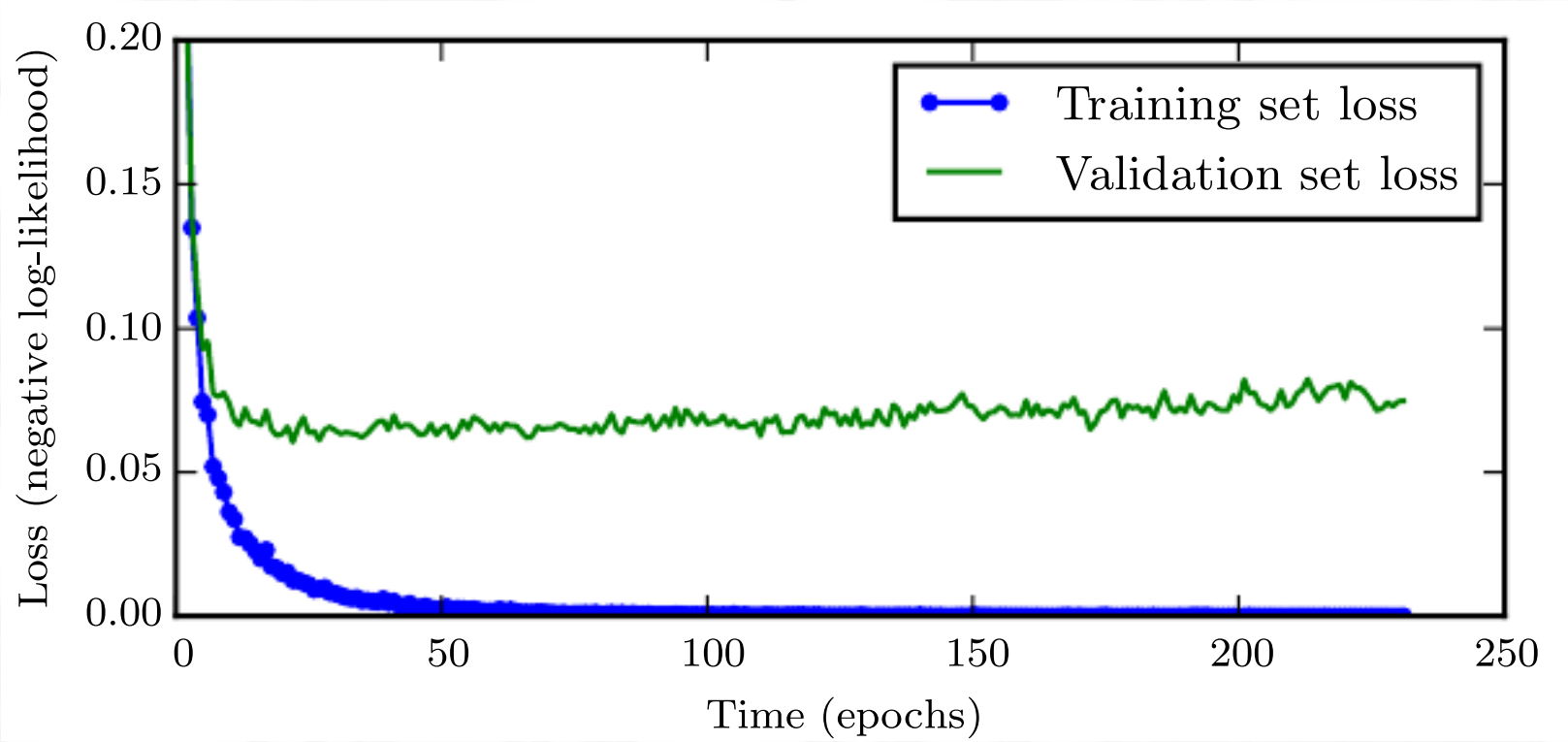
Regularization

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

Regularization – Early Stopping



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

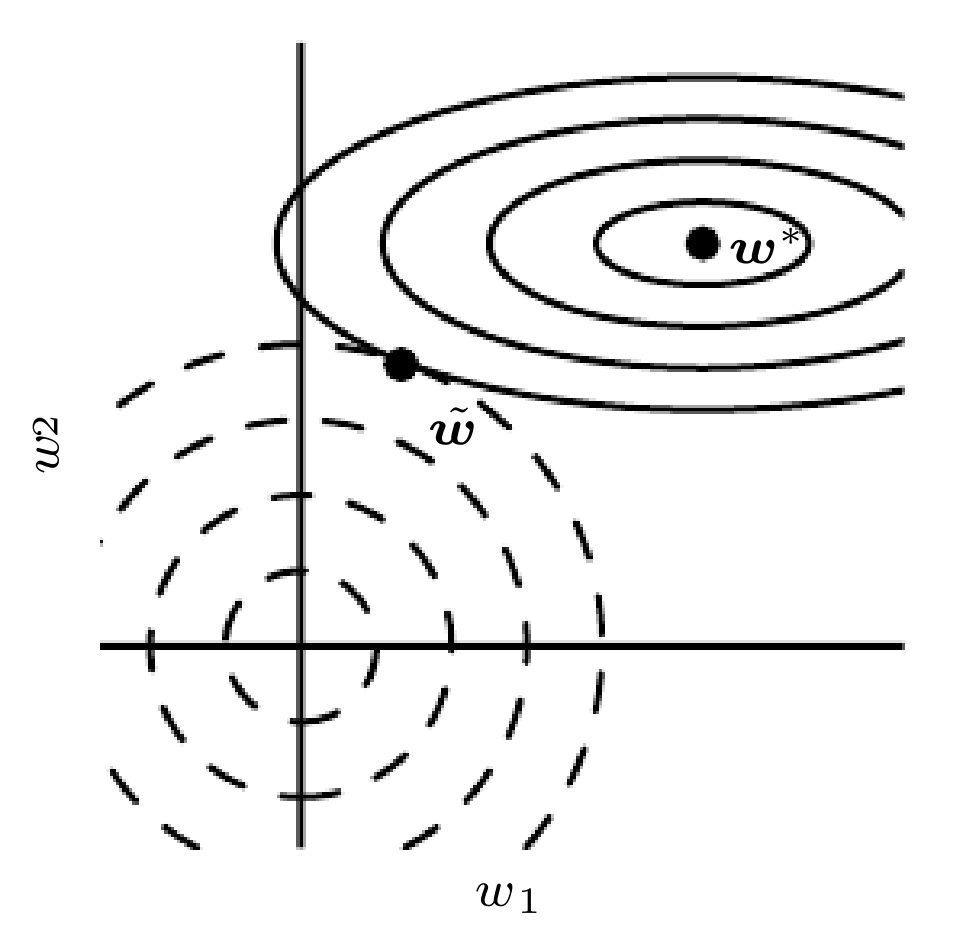
$$\tilde{J}(\boldsymbol{\theta}; \mathbb{X}) = J(\boldsymbol{\theta}; \mathbb{X}) + \lambda ||\boldsymbol{\theta}||_2^2$$

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



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 $\mathcal{N}(\boldsymbol{\theta}; 0, \sigma^2)$.

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

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

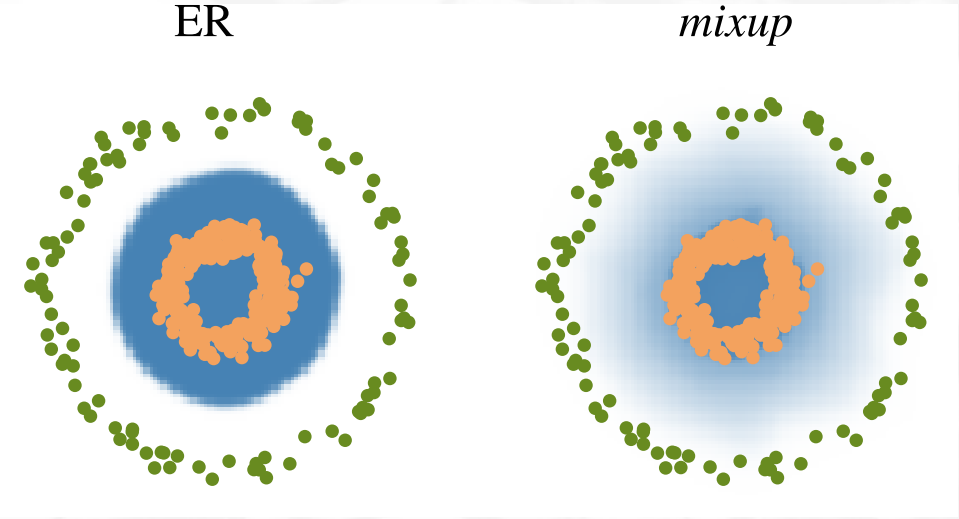
Regularization – Dataset Augmentation

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

- Image processing: translations, horizontal flips, scaling, rotations, color adjustments, …
- Speech recognition
- More difficult for discrete domains like text.

Regularization – Dataset Augmentation

- Noise injection to input examples
- Label smoothing
- Mixup (25 Oct 2017)



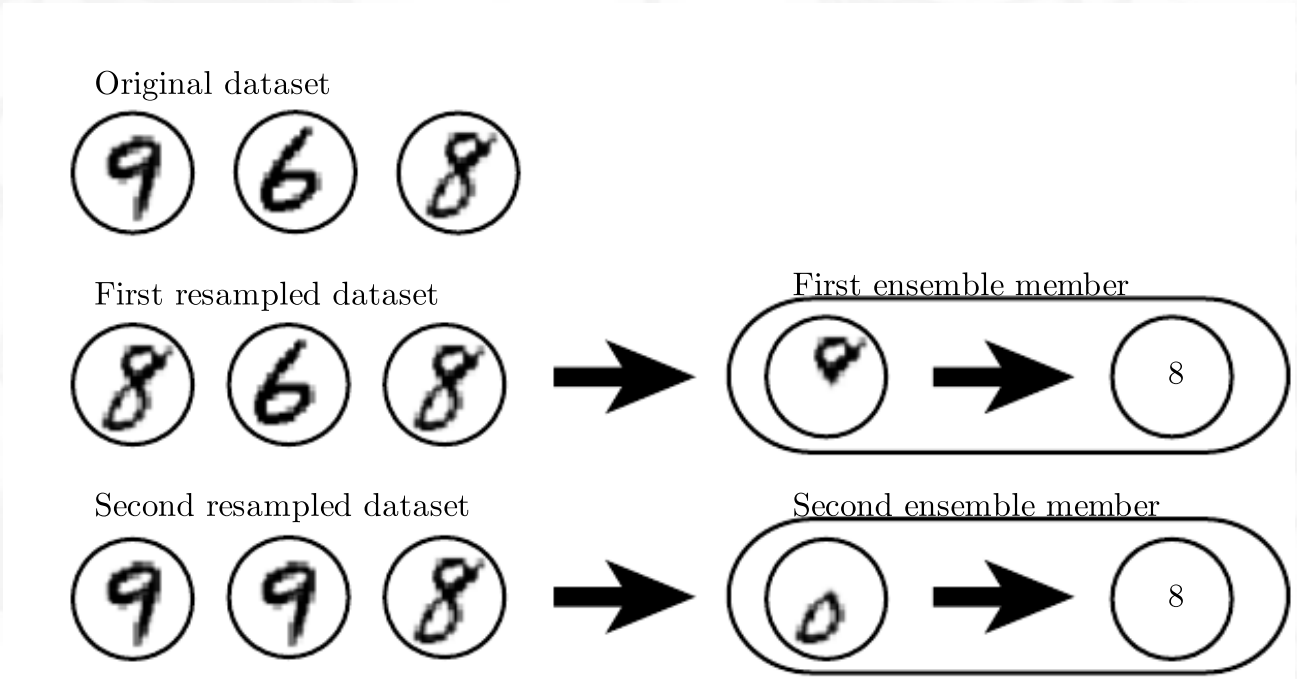
(b) Effect of *mixup* on a toy problem.

Regularization – Ensembling

Ensembling (also called *model averaging* or *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 regression).

Regularization – Ensembling

- Generate different datasets by sampling with replacement.



- 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.

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.

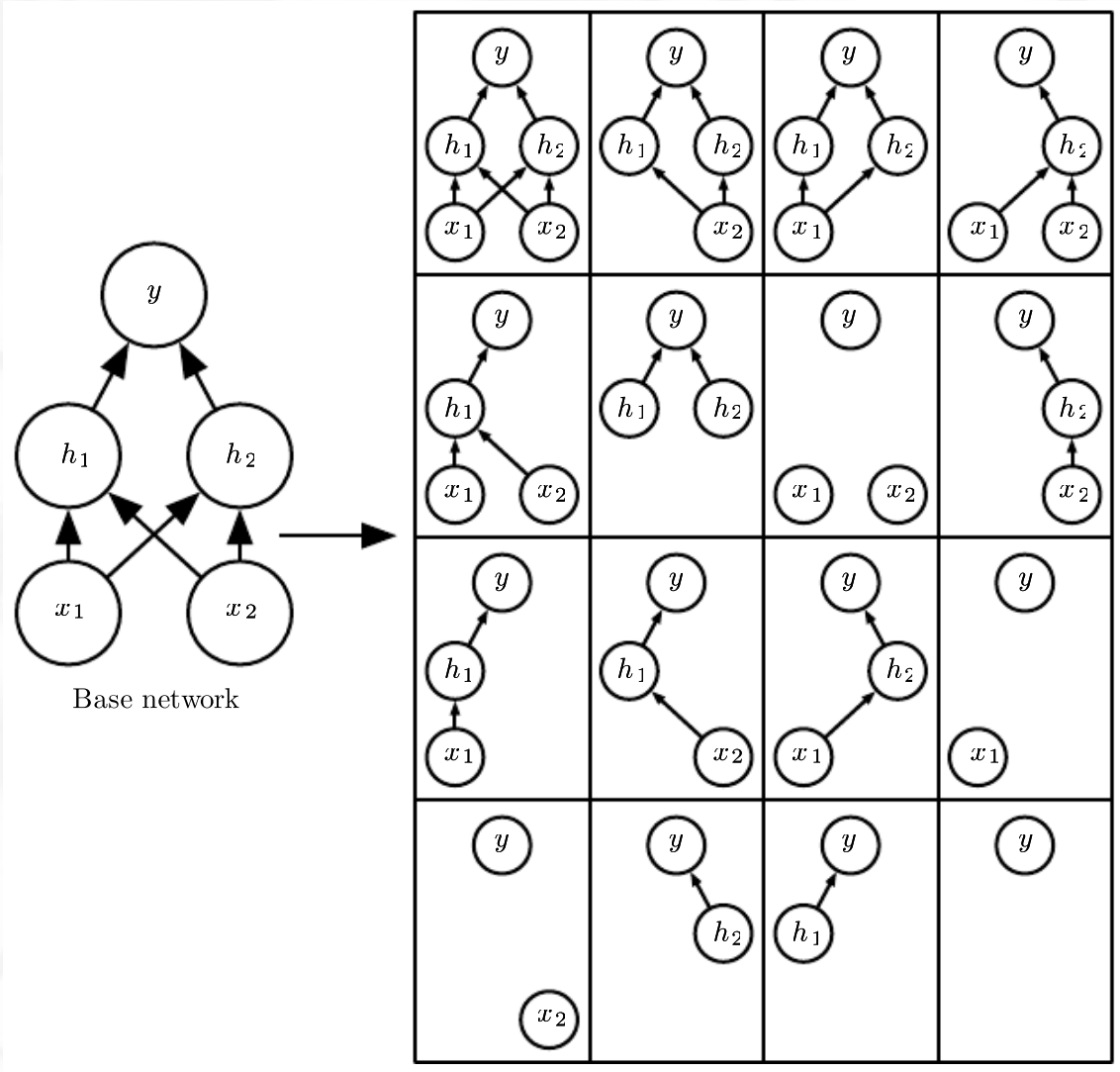
Idea of *dropout* by (Srivastava et al., 2014), in preprint since 2012.

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.

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.

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

Regularization – Dropout as Ensembling



Base network

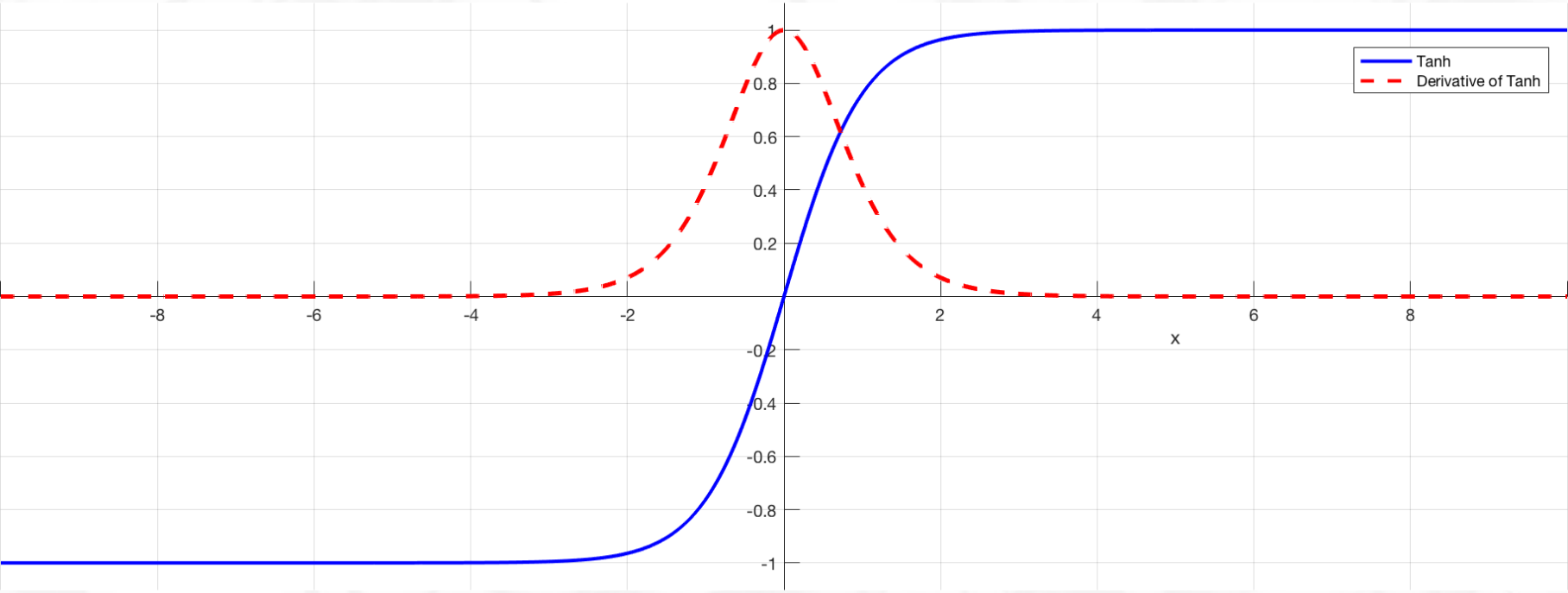
Ensemble of subnetworks

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.

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.

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.

Convergence – Parameter Initialization

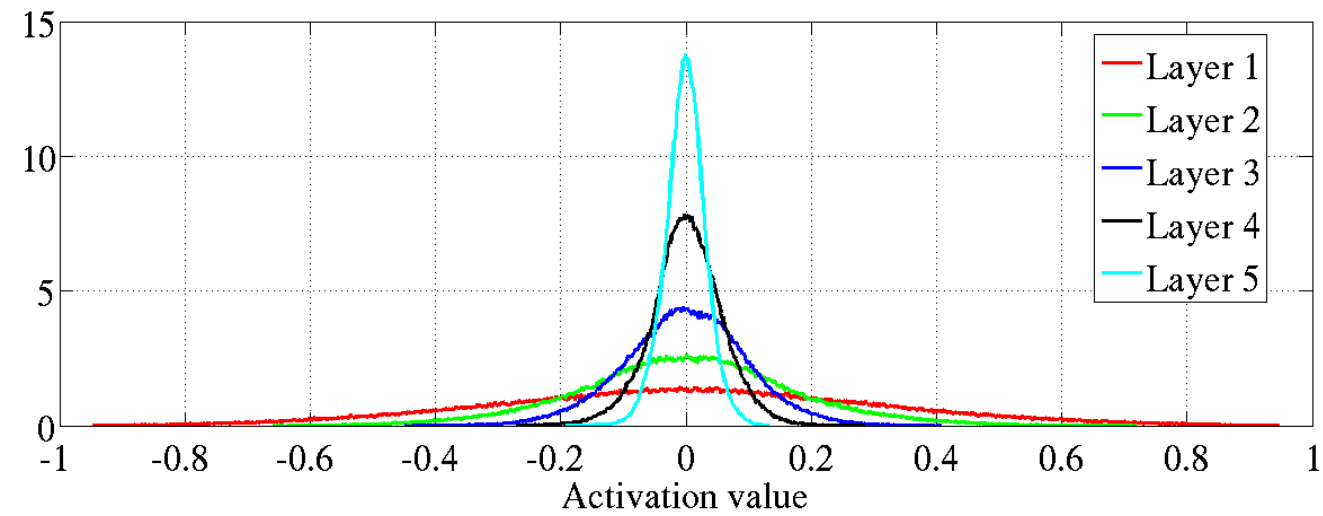
Neural networks usually need random initialization to *break symmetry*.

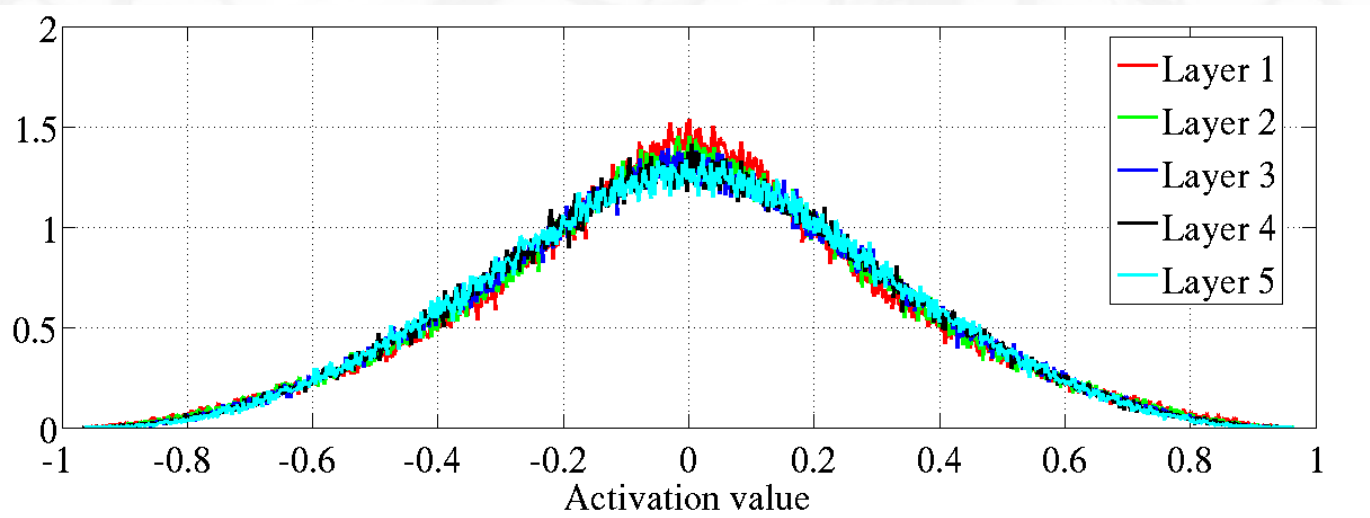
- Biases are usually initialized to a constant value, usually 0.
- Weights are usually initialized to small random values, either with uniform or normal distribution.
  - The scale matters for deep networks!
  - Originally, people used $U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$ distribution.
  - Xavier Glorot and Yoshua Bengio, 2010: *Understanding the difficulty of training deep feedforward neural networks*.

    Theoretically and experimentally showed that a suitable way to initialize a $\mathbb{R}^{n \times m}$ matrix is

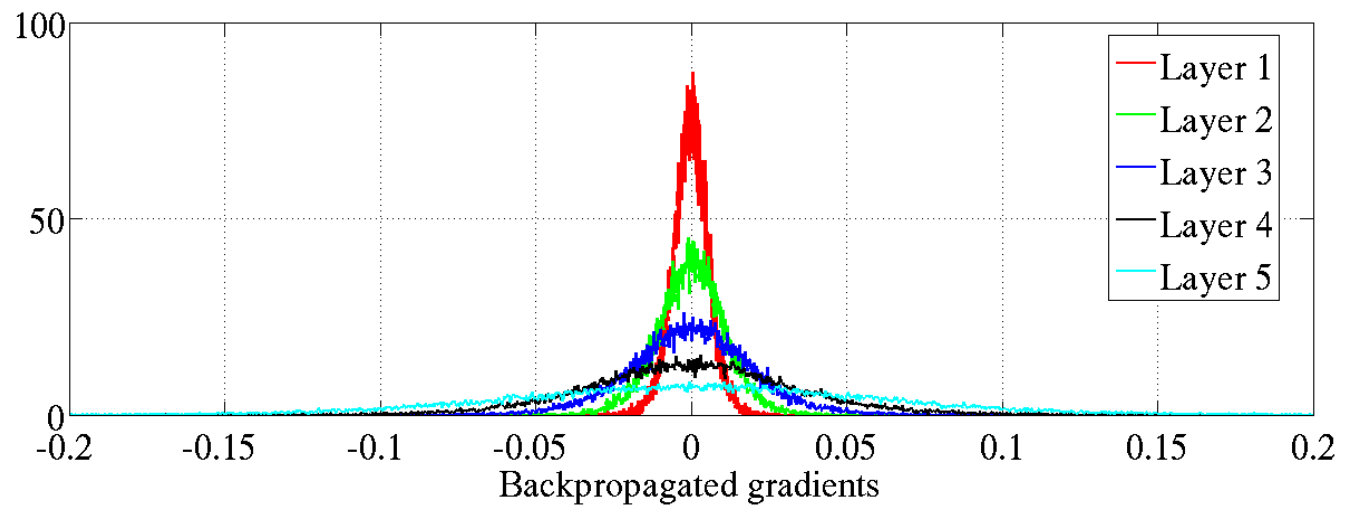
$$U\left[-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right].$$

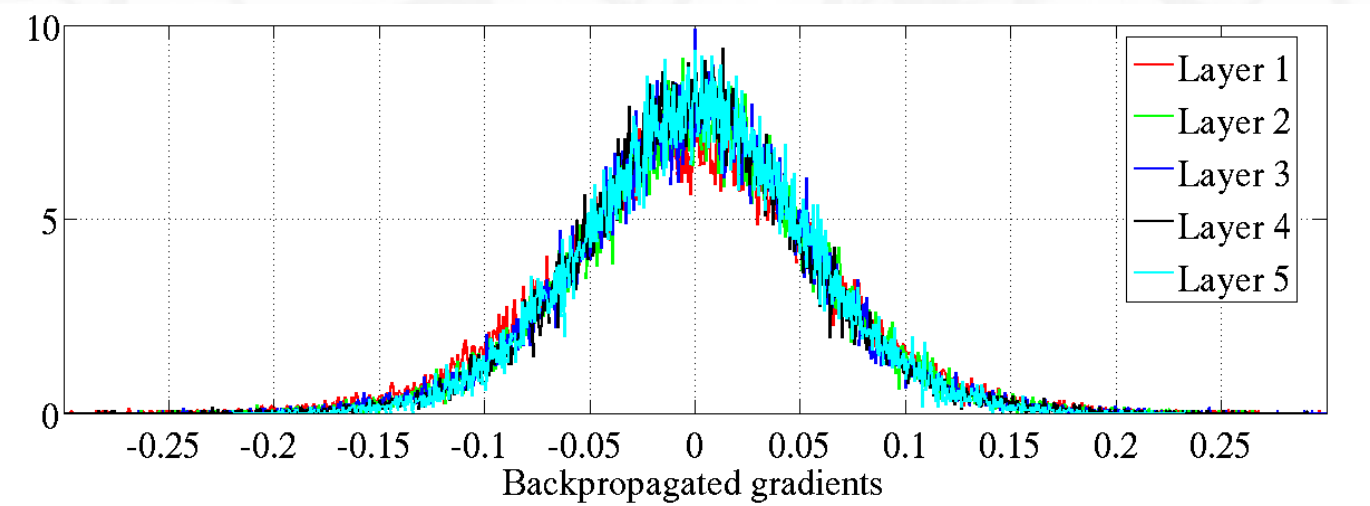
Convergence – Parameter Initialization



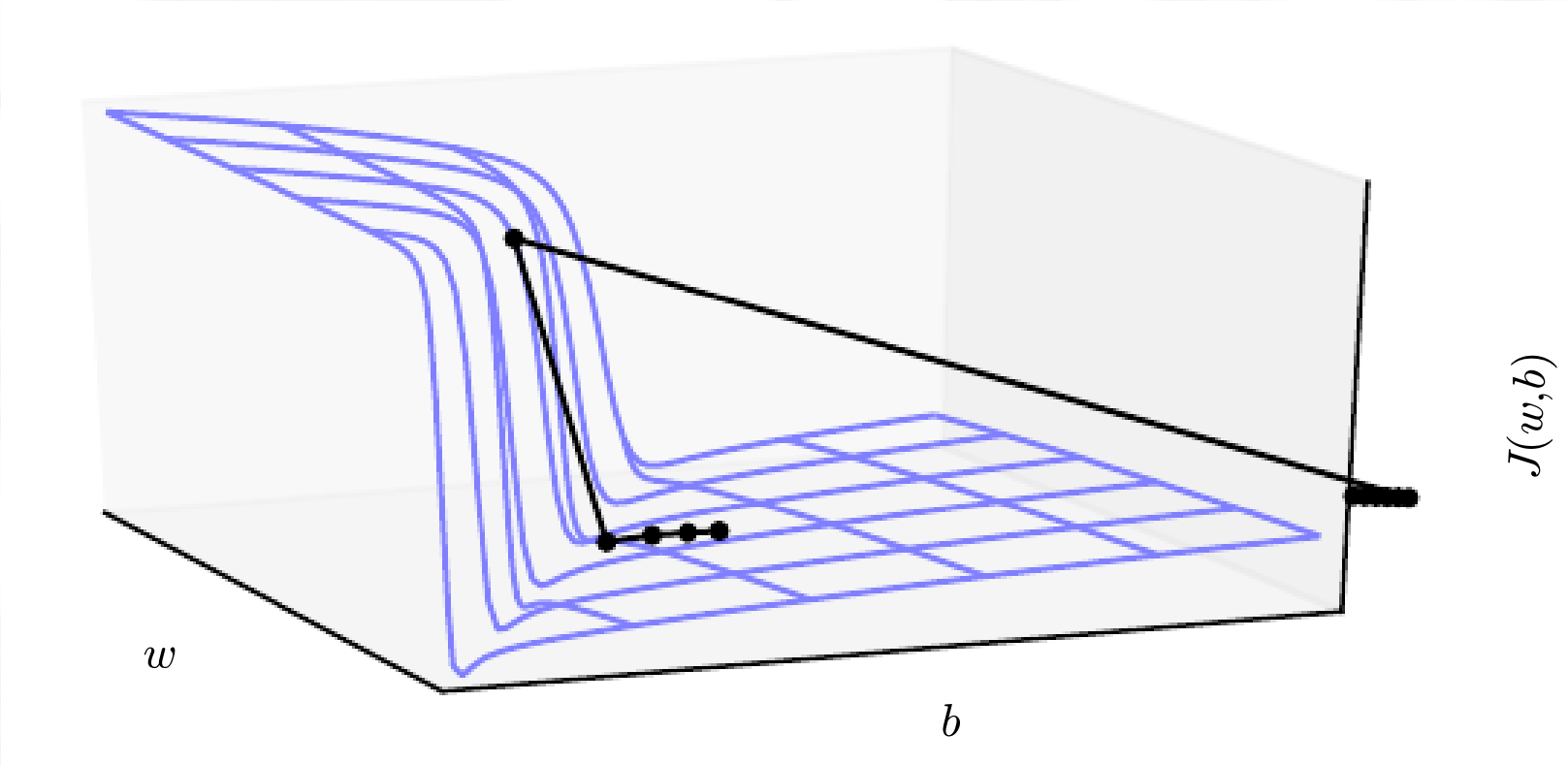


Convergence – Parameter Initialization

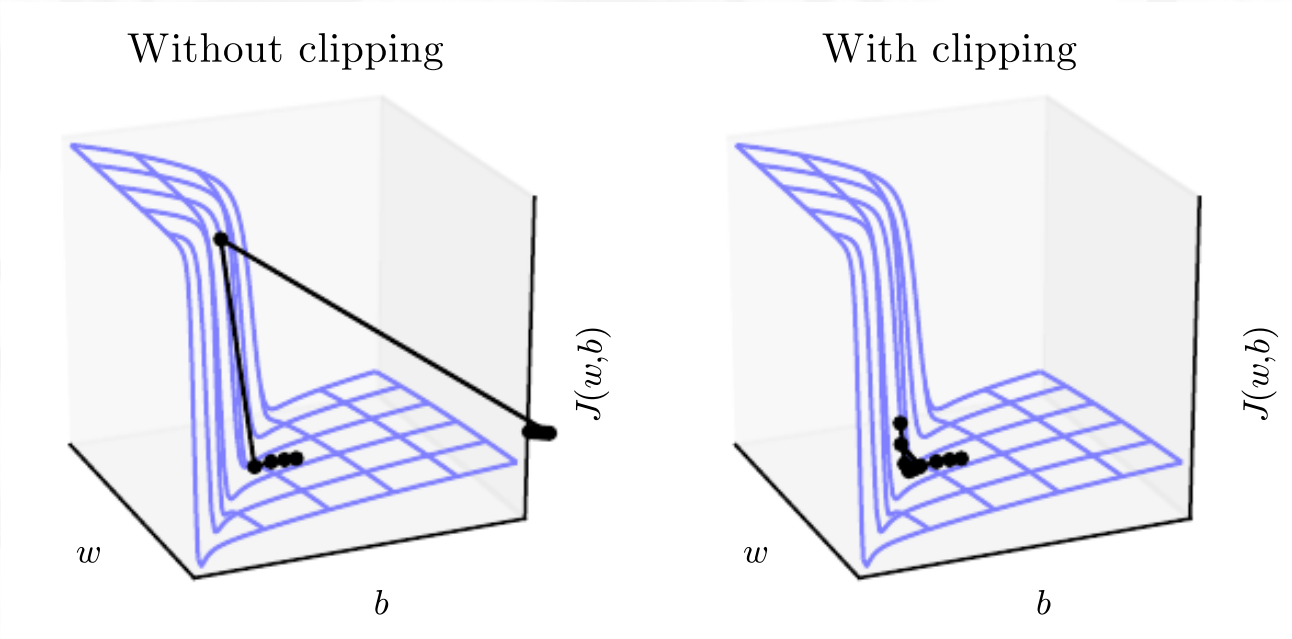




Convergence – Gradient Clipping



Convergence – Gradient Clipping



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, per matrix or for the gradient as a whole.

Going Deeper

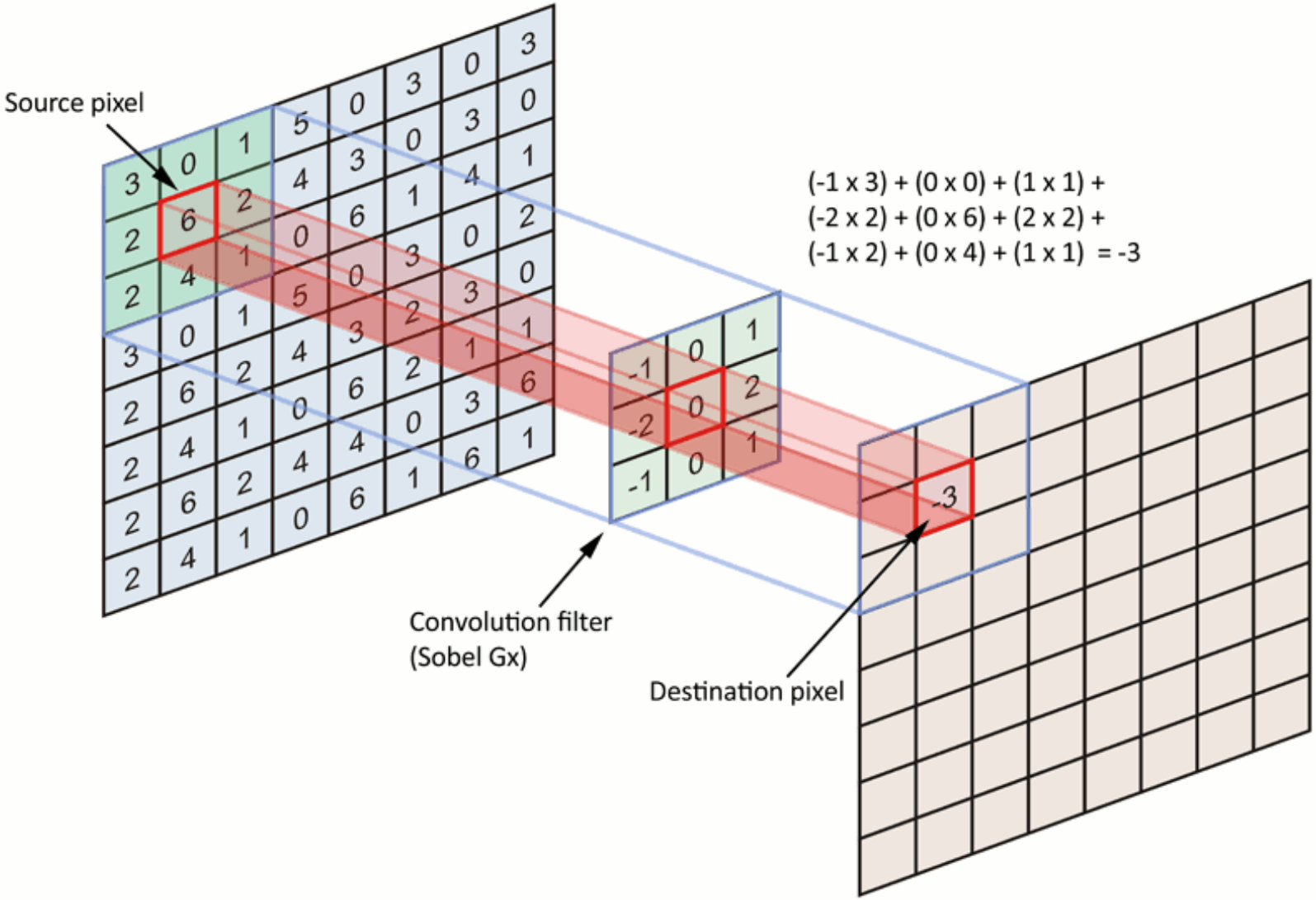
Convolutional Networks

Consider data with some structure (temporal data, speech, images, …).

Unlike densely connected layers, we might want:

- Sparse (local) interactions
- Parameter sharing (equal response everywhere)
- Shift invariance

Convolutional Networks



Convolution Operation

For a functions x and w, *convolution* $x * w$ is defined as

$$(x * w)(t) = \int x(a) w(t - a) \, \mathrm{d}a.$$

For vectors, we have

$$(\boldsymbol{x} * \boldsymbol{w})_t = \sum_i x_i w_{t-i}.$$

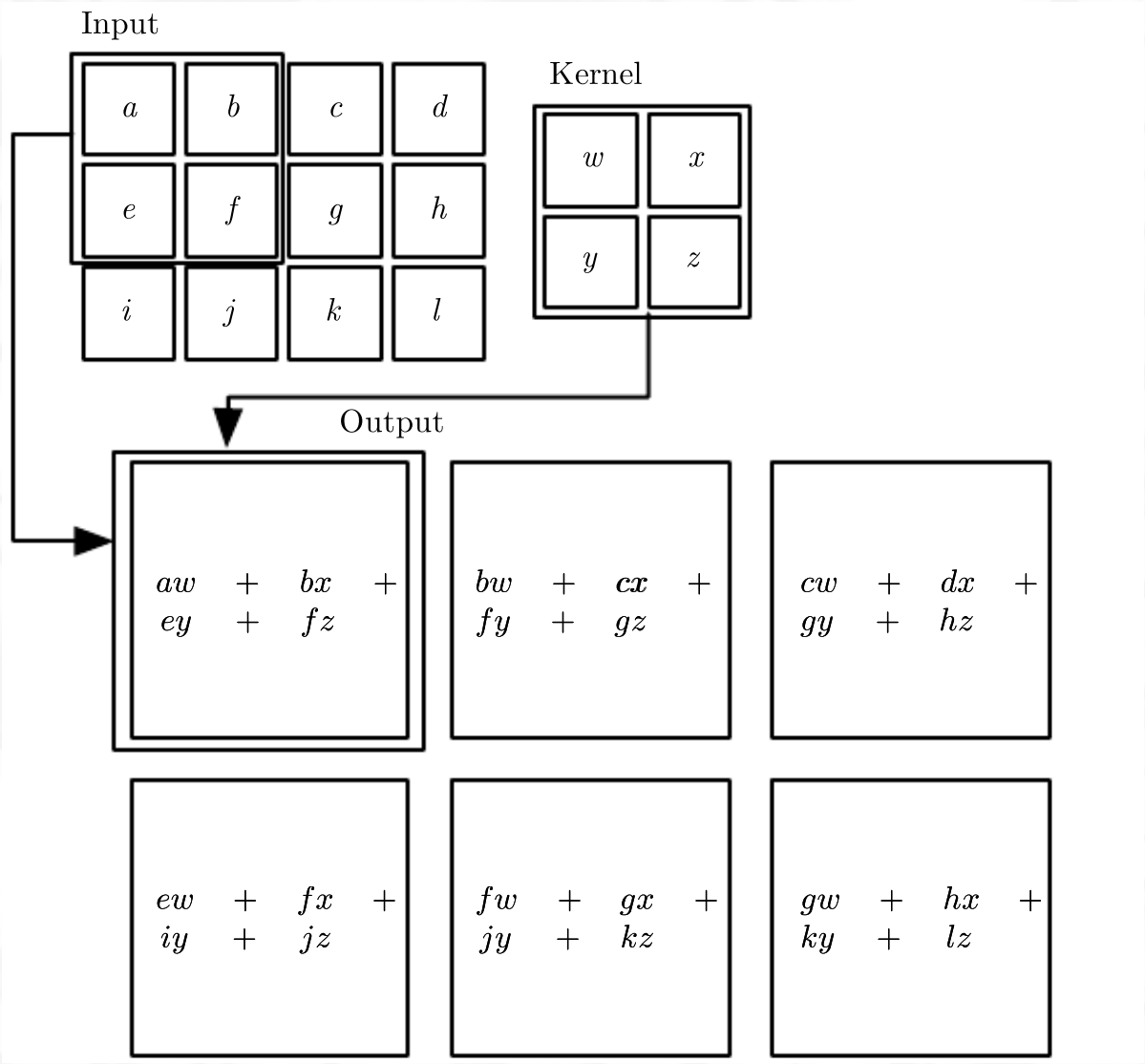
Convolution operation can be generalized to two dimensions by

$$(\boldsymbol{I} * \boldsymbol{K})_{i,j} = \sum_{m,n} \boldsymbol{I}_{m,n} \boldsymbol{K}_{i-m,j-n}.$$

Closely related is *cross-corellation*, where K is flipped:

$$S_{i,j} = \sum_{m,n} \boldsymbol{I}_{i+m,j+n} \boldsymbol{K}_{m,n}.$$

Convolution



Convolution Operation

The K is usually called a *kernel* or a *filter*, and we generally apply several of them at the same time.

Consider an input image with C channels. The convolution operation with F filters of width W, height H and stride S produces an output with F channels kernels of total size $W \times H \times C \times F$ and is computed as

$$(\boldsymbol{I} * \boldsymbol{K})_{i,j,k} = \sum_{m,n,o} \boldsymbol{I}_{i \cdot s + m, j \cdot s + n, o} \boldsymbol{K}_{m,n,o,k}.$$

Convolution Operation

The K is usually called a *kernel* or a *filter*, and we generally apply several of them at the same time.

Consider an input image with C channels. The convolution operation with F filters of width W, height H and stride S produces an output with F channels kernels of total size $W \times H \times C \times F$ and is computed as

$$(\boldsymbol{I} * \boldsymbol{K})_{i,j,k} = \sum_{m,n,o} \boldsymbol{I}_{i \cdot s + m, j \cdot s + n, o} \boldsymbol{K}_{m,n,o,k}.$$

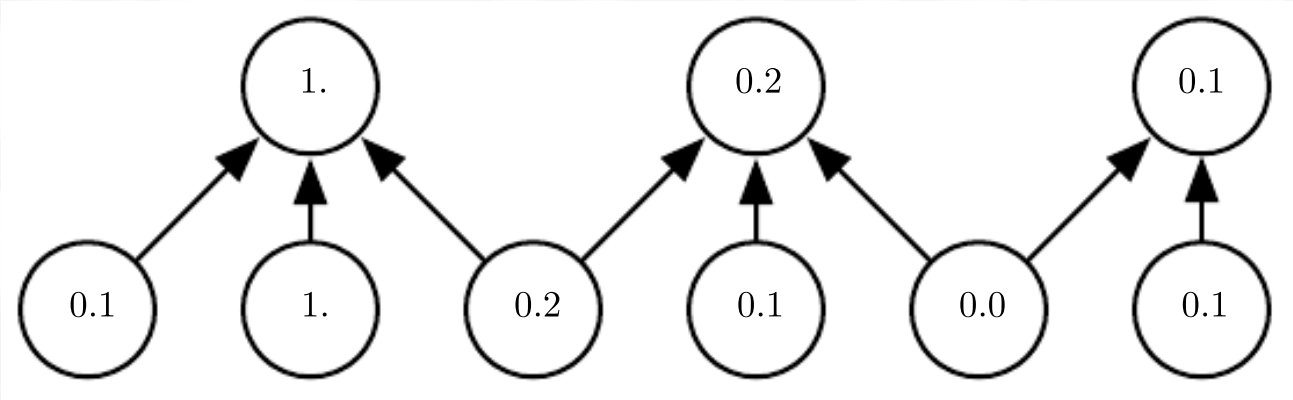
There are multiple padding schemes, most common are:

- valid: we only use valid pixels, which causes the result to me smaller
- same: we pad original image with zero pixels so that the result is exactly the size of the input

Pooling

Pooling is an operation similar to convolution, but we perform a fixed operation instead of multiplying by a kernel.

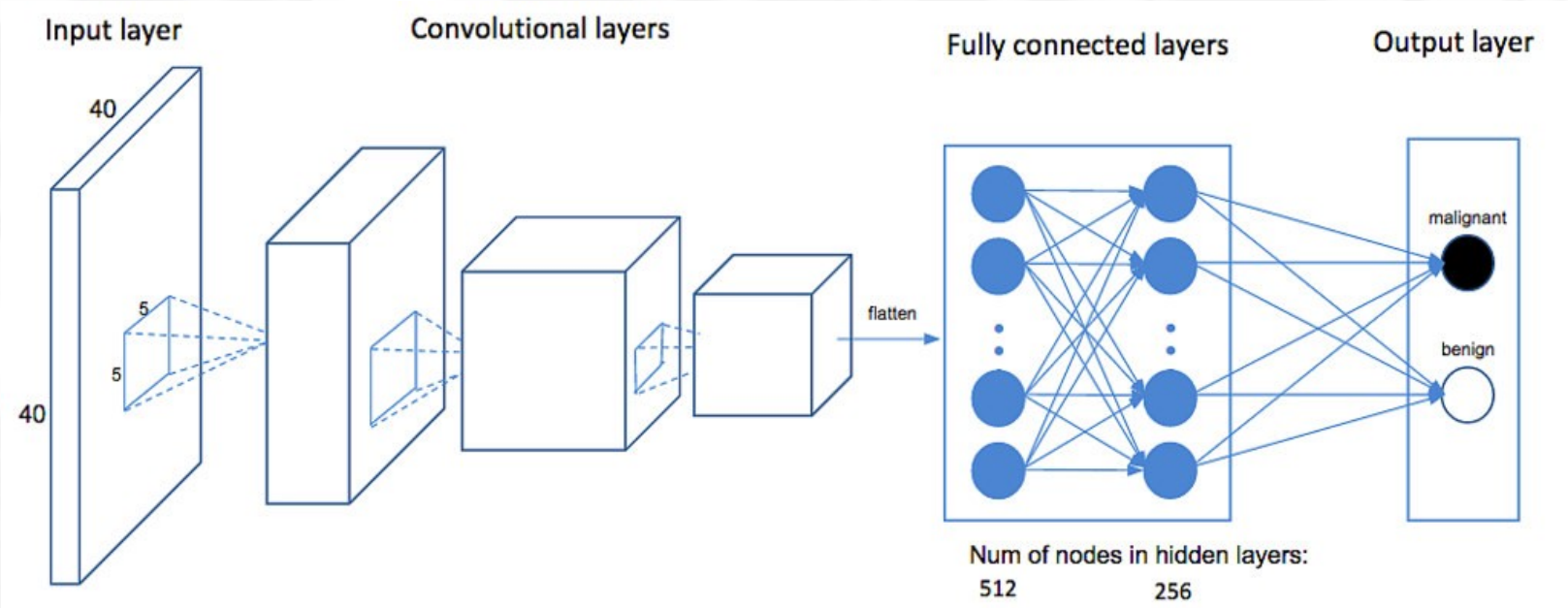
- Max pooling: minor translation invariance
- Average pooling



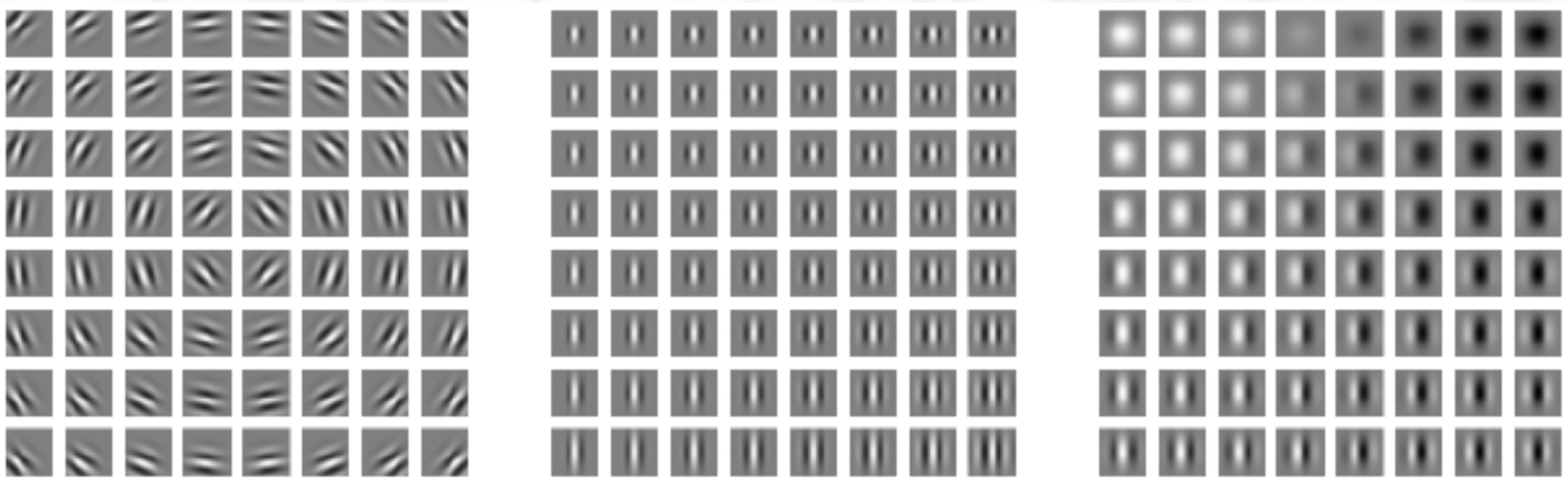
High-level CNN Architecture

We repeatedly use the following block:

1. Convolution operation
2. Non-linear activation (usually ReLU)
3. Pooling

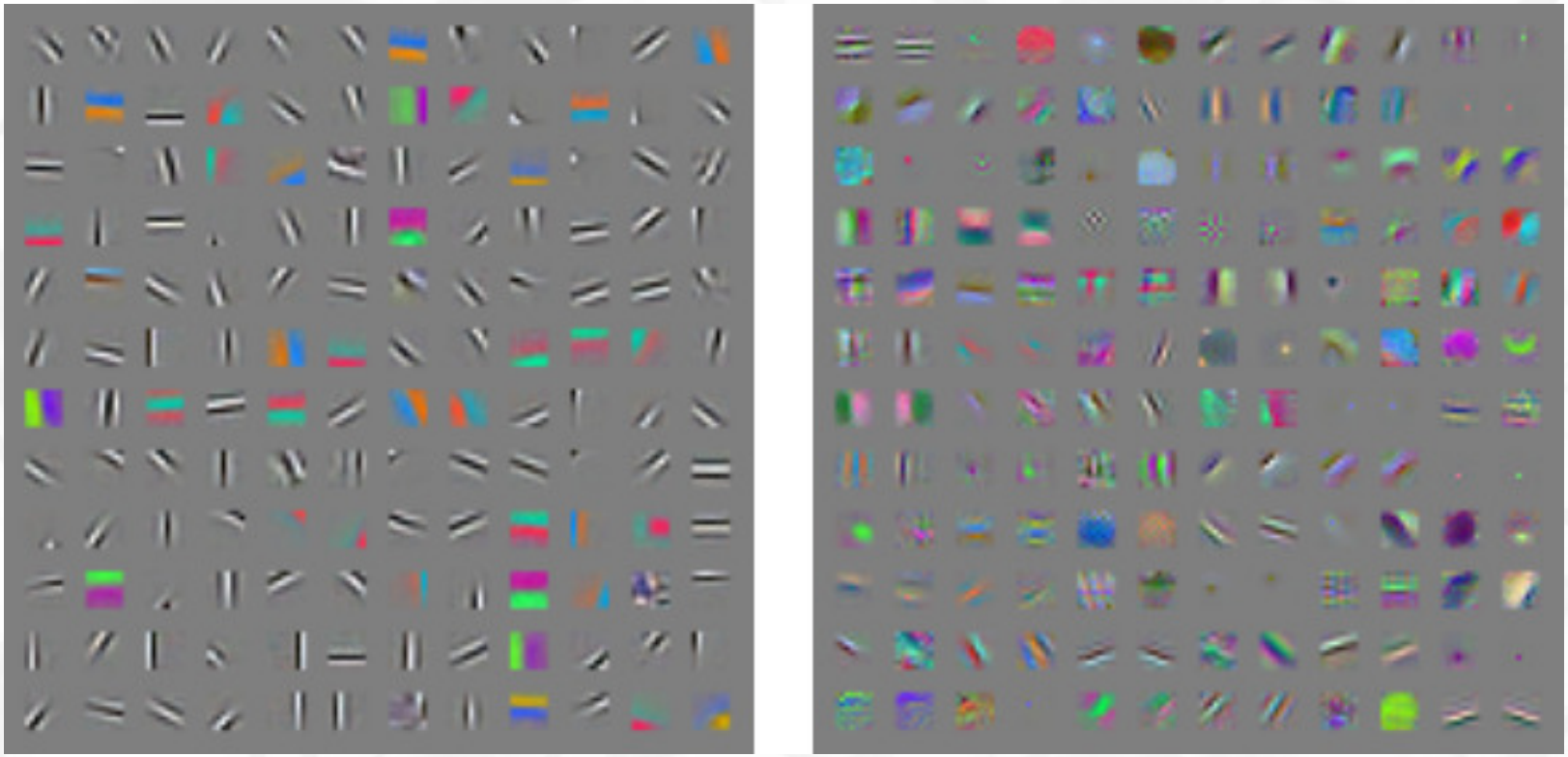


Similarities in V1 and CNNs



The primary visual cortex recognizes Gabor functions.

Similarities in V1 and CNNs



Similar functions are recognized in the first layer of a CNN.

AlexNet

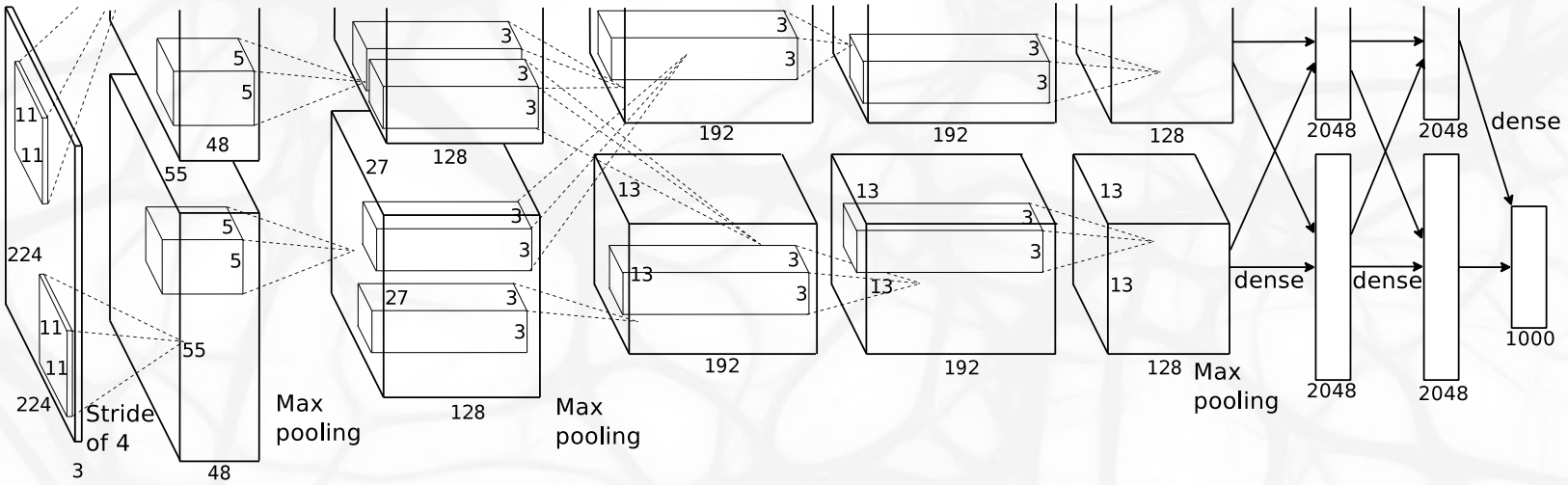


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253, 40–186,624–64,896–64,896–43,264–4096–4096–1000.

AlexNet