Biological Applications of Deep Learning
Lecture 5

Alexander Schönhuth

UNIVERSITÄT BIELEFELD
Faculty of Technology

Bielefeld University

November 9, 2022

Contents Today

- Reminder: Convolutional Neural Networks (CNNs)
- Getting CNNs to Work in Practice
- Choosing Hyperparameters
- Convolutional Backpropagation
- Training Variations

Convolutional Neural Networks (CNNs)
Reminder

Convolutional Neural Networks

Motivation

- Use that images have a spatial structure
  - ☞ Neighboring pixels are more likely to belong to the same structural elements
- Exploit this to speed up training, and reduce number of parameters (weights)

Basic Ideas

- Local receptive fields
- Shared weights
- Pooling

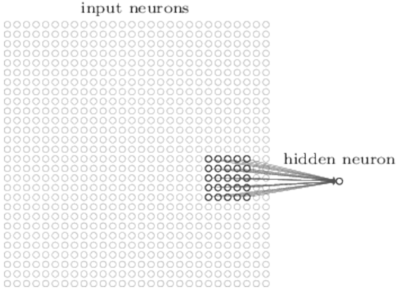
Local Receptive Fields and Convolutional Filters

Convolutional Neural Networks

Local Receptive Fields

In a convolutional NN,

- Every node in the first hidden layer is connected to a rectangular subregion
- Here: subregion = square of 5x5=25 input neurons

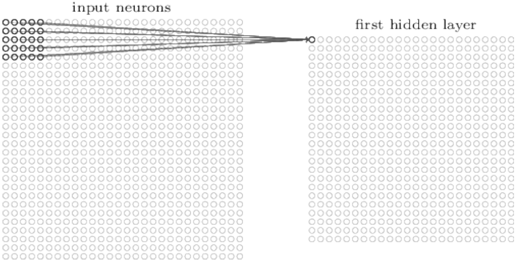


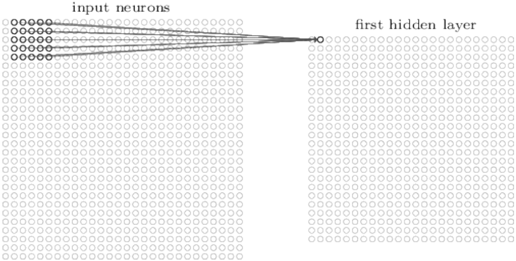
Convolutional filter of size 5 x 5

Definition
The region in the input images to which a hidden neuron is connected is called the *local receptive field (LRF)* of the hidden neuron.

Convolutional Neural Networks

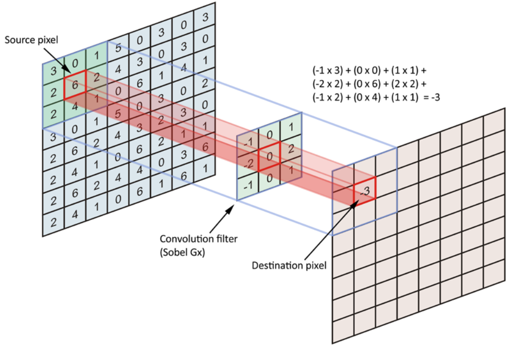
Local Receptive Fields





Convolutional Neural Networks

Convolutional Filters



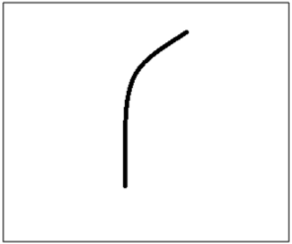
For generating one hidden layer, identical parameters, together defining one convolutional filter, are used

NEURAL NETWORKS

CONVOLUTION FILTERS

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



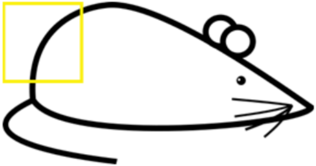
Visualization of a curve detector filter

Filter for recognizing a curve

Neural Networks

Convolution Filters

Original image



Visualization of the filter on the image

Neural Networks

Convolution Filters

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

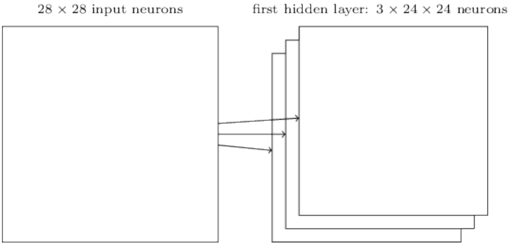
Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Convolutional Neural Networks

Convolutional Filters

Definition
A *feature map* is a mapping associated with one convolutional filter.



- A complete convolutional layer consists of several hidden sublayers
- Each sublayer is defined by one feature map

Sharing Weights

Convolutional Neural Networks

Shared Weights and Biases

- *Convolution Formula*: The activation a_{jk}^{l+1} of the j, k-th hidden neuron within the layer, using a $M \times M$ LRF, is computed as (σ may represent activation function of choice)

$$a_{jk}^{l+1} = \sigma(b + \sum_{l=0}^{M} \sum_{m=0}^{M} w_{l,m} a_{j+l,k+m}^{l}) \quad (1)$$

- *Observation:* For each node in the same hidden layer, the same parameters $w_{l,m}, 1 \leq l, m \leq M$ are used
- That is, we only need $M \times M$ parameters to generate the entire hidden layer

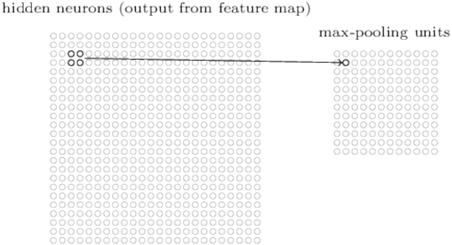
MNIST example

:

- Convolutional layer, 20 feature maps, each of size 5×5, roughly requires $20 \times 5 \times 5 = 500$ weights
- Fully connected network, connecting 784 input neurons with 30 hidden neurons requires $784 \times 30 = 23\,520$ weights
- **CNN requires roughly 40 times less parameters**

Pooling Layers

Convolutional Neural Networks
Pooling Layers

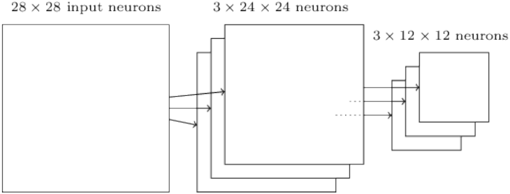


2×2 pooling

- *Max pooling*: Each $L \times L$ rectangle is mapped onto the maximum of its values
- *L2 pooling*: Each $L \times L$ rectangle is mapped to the rooted average of the squares of the values
- This overall yields a layer that is $L \times L$ times smaller
- Usually $L = 2$ is used

Convolutional Neural Networks

Combining Convolutional and Pooling Layers

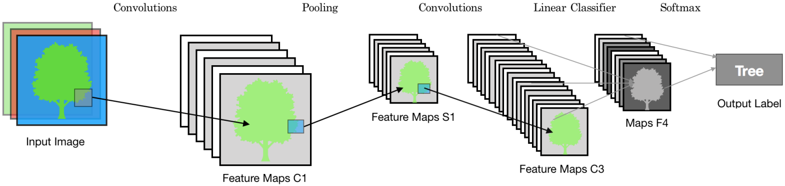


Convolutional layer followed by pooling layer

- Convolutional and pooling layers are used in combination
- Pooling layers usually follow convolutional layers
- *Intuition*:
  - The exact location of the occurrence of a feature is not important
  - Pooling helps to handle distortions and rotations

Convolutional Neural Networks

Architecture

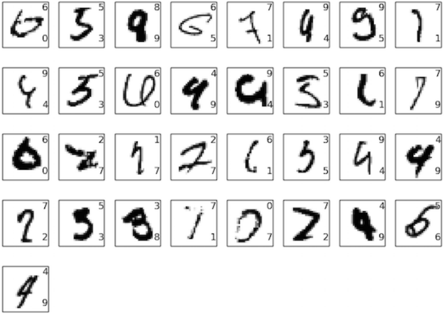


- Great depth at (relatively) little parameters
- Each filter recognizes substructure in image
- Substructures combine to larger structures ...
- ... until image can be classified

Convolutional Neural Networks in Practice

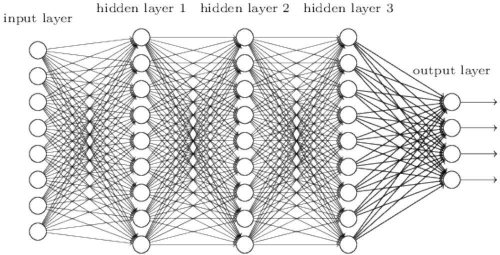
Goal

Setting up a neural network that correctly classifies 9967 out of 10 000 images; see below for the 33 misclassified ones.



33 misclassified images; correct/predicted classification upper/lower right corner

Fully Connected Networks



Fully connected neural network with 3 hidden layers

Issue: With fully connected NN's, we only reach about 98% accuracy in prediction.

Question: How to get to 99,67% accuracy?

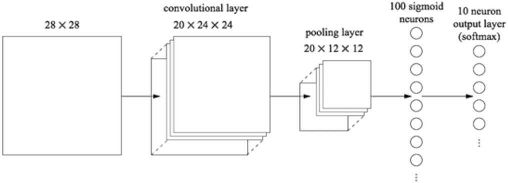
CNNs in Practice

Baseline: Simple Fully Connected Network

- *Baseline*:
  - One hidden layer, 100 neurons
  - Output layer, cost function: softmax + log-likelihood
- *Training*:
  - 60 epochs
  - Learning rate $\eta = 0.1$
  - Mini-batch size 10
- *Test accuracy*: 97.80%

CNNs in Practice

First CNN: one covolution-pooling layer



Inserting a convolution and max-pooling layer

- *Convolutional layer*:
  - 5×5 LRFs, stride length 1
  - 20 feature maps
- *Pooling layer*:
  - 2×2 max-pooling
- *Accuracy*: 98.78% test accuracy

CNNs in Practice

Two Convolution-Pooling Layers

- *2 Convolutional layers*:
  - *First convolution*: 20 feature maps, each associated with 5×5 LRFs, stride length 1
  - *Second convolution*: 40 feature maps, each associated with $20 \times 5 \times 5$ filter, stride length 1
- *Pooling layer*:
  - 2×2 max-pooling
- *Intuition*: after the first layer, each image consists of 12×12 pixels, where each pixel has 20 channels, each of which codes for a different "color"
- So, each LRF corresponds to $20 \times 12 \times 12$ tensor
- Spatial strucure is still preserved in second conv-pooling layer, so employing conv-pooling makes sense
- *Accuracy*: 99.06% test accuracy

CNNs in Practice

Trying alternative activation functions

- *Tanh activation function*:
  - *Definition*:
$$\sigma(z) = \frac{1 + \tanh(z/2)}{2} \tag{2}$$
  - Training is (a bit) faster
  - Results are near-identical
- *Rectified linear units (ReLUs)*:
  - Activation:
$$f(z) = \max(0, z)$$
  - Learning rate: $\eta = 0.03$ (earlier: 0.1)
  - L2 regularization at $\lambda = 0.1$
  - *Test accuracy*: 99.23%
  - Modest gain, but also in other experiments ReLUs have shown to consistently outperform sigmoid neurons

CNNs in Practice

Expanding the Training Data

- *Experiment*:
  - Displace each image by one pixel to above, the right, below, or to the left
  - Each image has 4 extra copies
  - ☞ 250 000 images instead of 50 000
- Run the same network with ReLU's (99.23%)
- Expanding training data yields 99.37%
- P. Simard, D. Steinkraus, J. Platt, "Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis", 2003:
  - Very similar architecture
  - Training data expansion: rotations, translations, skewing
  - "Elastic distortions": emulating random oscillations of hand muscles
  - Accuracy: 99.6%

CNNs in Practice

Extra/Larger Fully Connected Layer

- *Larger fully connected layer*:
  - 300 neurons ☞ accuracy: 99.46%
  - 1000 neurons ☞ accuracy: 99.43%
  - Not really convincing
- *Extra fully connected layer*:
  - 2 fully connected layers, each of 100 neurons ☞ accuracy 99.43%
  - 2 fully connected layers, each of 300/1000 neurons ☞ accuracy 99.47/99.48%
- No convincing improvements

CNNs in Practice

Dropout

- 2 fully connected layers each of 1000 neurons
- *Dropout* (probability = 0.5) applied to neurons in fully connected layers
- Accuracy: 99.6% (which is substantial improvement)
- *Remarks*:
  - Less epochs (40 instead of 60), because of faster training
  - More hidden neurons (1000 instead of 300 or 100) slightly preferable when using dropout
  - No dropout on convolutional layers: those have in-built resistance to overfitting because of parameter sharing

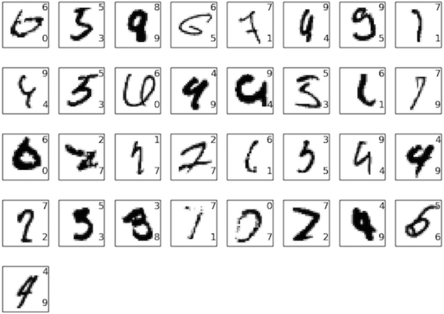
Ensemble of networks: Idea

- Train several different networks
- For example, employ repeated random initialization while always using the same architecture
- For classification, take the majority vote of the different networks
- While each network performs similarly, the majority vote may yield improvements
- Here: 5 randomly initialized network of the architecture o described in the slides before
- Accuracy: 99.67%
- That has been our goal!

CNNs in Practice

Ensemble of Networks

- Ensemble of 5 randomly initialized networks
- Architecture as described in the slides before
- Accuracy: 99.67% – that has been our goal!



33 misclassified images; correct/predicted classification upper/lower right corner

CNNs in Practice

References

- Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition", http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf [Architecture: "LeNet-5"]

CNNs on MNIST

Further Improvements

- For further improvements on MNIST (and on famous datasets in general see `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`
- *Noteworthy*:
  - See D.C. Ciresan, U. Meier, L.M. Gambardella, J. Schmidhuber, "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition", `https://arxiv.org/abs/1003.0358`
  - Fully connected network, without convolutional layers that achieves 99.65% accuracy.
  - Training for that non-convolutional network proceeds very slow, however.

Initializing Weights

Weight Initialization

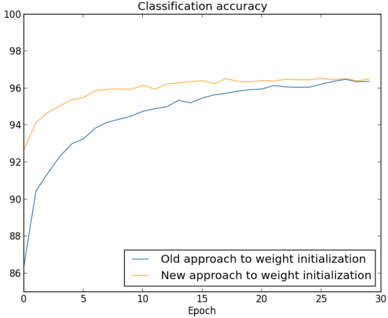
- Draw weights from normal distribution $\mathcal{N}_{0,\sigma}$ with mean 0 and standard deviation σ
- Let n_{in} be the number of inputs to the next node:

$$z = \sum_{j=1}^{n_{\text{in}}} x_j w_j + b \tag{3}$$

- Then $\sigma := \sqrt{n_{\text{in}}}$, so sample weights from $\mathcal{N}_{0,\sqrt{n_{\text{in}}}}$
- *Explanation*: So, input z to next node will (roughly) be sampled from $\mathcal{N}_{0,1}$

Weight Initialization

Consequence



- Improved initialization leads to faster learning
- See further [“Practical Recommendations for Gradient-Based Training of Deep Architectures”, Bengio, 2012] (`https://arxiv.org/pdf/1206.5533v2.pdf`) for more details.

Choosing Hyperparameters

Choosing Hyperparameters

Hyperparameters to be determined:

- Number (and composition) of hidden neurons
- In stochastic gradient descent: mini-batch size
- Number of epochs in training
- Learning rate η
- Regularization parameter λ
- If chosen inappropriately
  ☞ random exploration of search space
  ☞ no training will take place

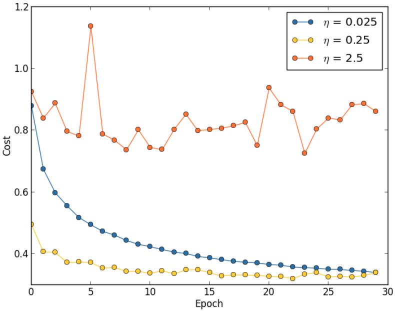
First Challenge

Establish *any* non-trivial learning, that is, train a network that classifies better than chance

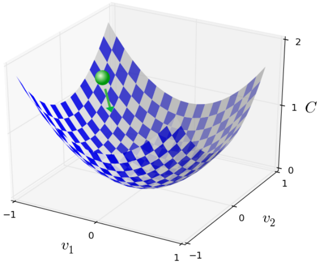
Strategies

- Turn multi-class problem into binary problem
- Start experimenting with the simplest possible architecture
- Small batch size: monitor changes in classification accuracy after each batch
- Check that *weight decay* (see (26), Lecture 3) is constant with respect to number of training data ☞ affects both η and λ

Choosing Learning Rate



Choosing Learning Rate



- Learning rate η too large: random oscillations ☞ parameters "jump across" optimum, back and forth
- Learning rate η too small: training too slow
- *Strategy*: Pick η as large as possible, while avoiding random oscillation
- To refine training, decrease η along epochs

Choosing Regularization Parameter

Suggestions

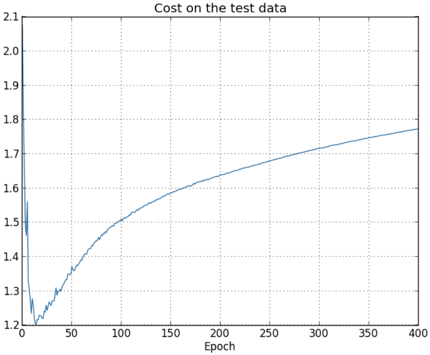
- Start with no regularization ($\lambda = 0$)
- Determine the learning rate η, as described above
- Then do $\lambda = 1.0$, and compare accuracy with $\lambda = 0$
- Depending on the outcome, multiply or divide by ten ($\lambda = 10.0$ or $\lambda = 0.1$)
- Once reached the right order of magnitude, finetune

Choosing Number of Epochs

- Use validation data; see earlier lectures for training, validation and test data. Validation is to be used for determining hyperparameters.
- Stop as soon as validation accuracy, the ratio of correctly classified validation data samples over the total number of validation data samples, no longer improves
- "No improvement-in-ten rule": stop 10 epochs after classification accuracy starts to stall

Choosing Number of Epochs

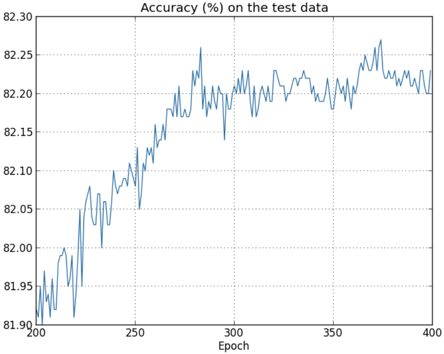
Cost versus Validation Accuracy



Validation accuracy (here: test accuracy) suggests to do $\approx$ 280 epochs

Choosing Number of Epochs

Cost versus Validation Accuracy



Validation cost (here: test cost) suggests to do ≈ 15 epochs

- Use cost or validation accuracy to determine number of epochs?
- Cost has no meaning, while accuracy does: use accuracy!

Choosing Mini-Batch Size

Prelude

See “Fully matrix-based approach to backpropagation over a mini-batch” in `http://neuralnetworksanddeeplearning.com/chap2.html`:

- Compute the gradient for each training datum separately
  - Can be done in parallel
- Average gradients across training data
- *Advantage:* Computation requires only half the time
- *Disadvantage:* One training datum versus batch of size m:

$$w \leftarrow w - \eta \nabla_w C_x \quad \text{versus} \quad w \leftarrow w - \eta \frac{1}{m} \sum_x \nabla_w C_x \tag{4}$$

  Updates per training datum small ☞ slow learning!
- Anything to do about this trade-off?

Choosing Mini-Batch Size

Solution

- *Observation*: Multiplying η by m yields

$$w \leftarrow w - \eta \sum_x \nabla_w C_x \tag{5}$$

which looks like summing over all individual examples, so issue of too little, and too small updates when using mini-batches mended.

Summary

- *Mini-batch size too small*: One does not exploit the advantages of matrix computation libraries.
- *Mini-batch size too large*: Too little updates.
- *Overall solution*: Find a good trade-off!
- Mini-batch size is fairly independent of other parameters.
- So, first optimize other hyperparameters. Then tune mini-batch size scaling η according to (5).

Choosing Hyperparameters

Search Techniques

- *Grid Search*: Try combinations of hyperparameters, viewing them as points of a grid, where each dimension refers to one of the hyperparameters
  - See `http://www.deeplearningbook.org/` 11.4.3 for details
- *Random Search*: Randomly pick combinations of hyperparameters, selected according to reasonable probability distributions
  - See `http://www.deeplearningbook.org/` 11.4.4 for details
- *Model-Based Hyperparameter Optimization*: Cast selection of hyperparameters as optimization problem, and try to pick hyperparameters that yield minimal error on validation data
  - See `http://www.deeplearningbook.org/` 11.4.5 for details
  - And the following slides for further information on automated optimization strategies

Automated Techniques

- ["Random search for hyper-parameter optimization", Bergstra & Bengio, 2012; https://dl.acm.org/citation.cfm?id=2188395]
- ["Practical Bayesian optimization of machine learning algorithms", Snoek, Larochelle & Adams, 2012; http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf]

Choosing Hyperparameters

Guidelines for Automated Techniques

Automated Techniques

- Also possible: "Auto Machine Learning (AutoML)", methods to pick optimal selections of hyperparameters, in particular to pick optimal network architectures.
- See
  - `https://hackernoon.com/a-brief-overview-of-automatic-machine-learning-solutions-automl-2826c7807a2a`
  - `https://ai.googleblog.com/2017/05/using-machine-learning-to-explore.html`

  if interested
- *However*: usually very expensive in terms of compute resources

Practical Recommendations

Further Reading

- "Practical Recommendations for gradient-based training of deep architectures", Y. Bengio, 2012, see `https://arxiv.org/abs/1206.5533`
- "Efficient BackProp", Y. LeCun, L. Bottou, G. Orr, K.-R. Müller, 1998, see `http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf`
- "Neural Networks: Tricks of the Trade", edited by G. Montavon, G. Orr, K.-R. Müller, see `https://www.springer.com/de/book/9783642352881`
  This book contains the above articles, is expensive, but many of the articles that appear in the book are freely available

Backpropagation for CNNs

Convolutional Backpropagation

Notation

- In fully connected layers we had [w^l_{jk} is for connecting the k-th neuron in layer $l-1$ with the j-th neuron in layer l]

$$\delta^l_j = \frac{\partial C}{\partial z^l_j} \quad \text{where} \quad z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j \tag{6}$$

  and $a^l_j = \sigma(z^l_j)$ (where σ is any activation function)
- For sake of simplicity, assume only one hidden sublayer
  - corresponds to one feature map, or one channel
  - only one convolutional filter per level of depth required

Convolutional Backpropagation

Notation

- We index neurons using two coordinates, so $z^l_{x,y}$ is the input for the x, y-th neuron of the hidden layer at level of depth l.
- The $M \times M$ filter that connects neurons from level l with neurons at level $l+1$ has weights $w^{l+1}_{ab}, 1 \leq a, b \leq M$
- By applying the convolution operation [and neglecting the exact indexing in the following]

$$z^{l+1}_{x,y} = \sum_a \sum_b w^{l+1}_{ab} \sigma(z^l_{x-a,y-b}) + b^{l+1}_{x,y} \tag{7}$$

Convolutional Backpropagation

- We compute

$$\delta^l_{x,y} = \frac{\partial C}{\partial z^l_{x,y}} = \sum_{x'} \sum_{y'} \frac{\partial C}{\partial z^{l+1}_{x',y'}} \frac{\partial z^{l+1}_{x',y'}}{\partial z^l_{x,y}} \tag{8}$$

- Moving on, we get

$$\begin{aligned} \frac{\partial C}{\partial z^l_{x,y}} &= \sum_{x'} \sum_{y'} \frac{\partial C}{\partial z^{l+1}_{x',y'}} \frac{\partial z^{l+1}_{x',y'}}{\partial z^l_{x,y}} \\ &= \sum_{x'} \sum_{y'} \delta^{l+1}_{x',y'} \frac{\partial(\sum_a \sum_b w^{l+1}_{ab} \sigma(z^l_{x'-a,y'-b}) + b^{l+1}_{x',y'})}{\partial z^l_{x,y}} \end{aligned} \tag{9}$$

Convolutional Backpropagation

- All terms in (9) where $x \neq x' - a$ or $y \neq y' - b$ are zero, so

$$\sum_{x'} \sum_{y'} \delta^{l+1}_{x',y'} \frac{\partial(\sum_a \sum_b w^{l+1}_{ab} \sigma(z^l_{x'-a,y'-b}) + b^{l+1}_{x',y'})}{\partial z^l_{x,y}} = \sum_{x'} \sum_{y'} \delta^{l+1}_{x',y'} w^{l+1}_{ab} \sigma'(z^l_{x,y}) \quad (10)$$

- Since now $x = x' - a$, $y = y' - b$, we have $a = x' - x$, $b = y' - y$, so

$$\sum_{x'} \sum_{y'} \delta^{l+1}_{x',y'} w^{l+1}_{ab} \sigma'(z^l_{x,y}) = \sum_{x'} \sum_{y'} \delta^{l+1}_{x',y'} w^{l+1}_{x'-x,y'-y} \sigma'(z^l_{x,y}) \quad (11)$$

Convolutional Backpropagation

- Summary:

$$\delta^l_{x,y} = \sum_{x'} \sum_{y'} \delta^{l+1}_{x',y'} w^{l+1}_{x'-x,y'-y} \sigma'(z^l_{x,y}) \tag{12}$$

- A closer look reveals this as a convolution operation in its own right, applying the filter

$$\sigma'(z^l_{x,y}) \cdot \begin{pmatrix} w_{MM} & \cdots & w_{M1} \\ \vdots & \ddots & \vdots \\ w_{1M} & \cdots & w_{11} \end{pmatrix} \tag{13}$$

  to the $l+1$-*th layer of gradients* $\delta^{l+1}_{x',y'}$, for computing values $\delta^l_{x,y}$ of the l-th layer of gradients.

Convolutional Backpropagation

- *Convolutional backpropagation:* applying filter

$$\sigma'(z^l_{x,y}) \cdot \begin{pmatrix} w_{MM} & \cdots & w_{M1} \\ \vdots & \ddots & \vdots \\ w_{1M} & \cdots & w_{11} \end{pmatrix} \tag{14}$$

  to the $l+1$-*th layer of gradients* $\delta^{l+1}_{x',y'}$
- ☞ Weights of original filter have been rotated by 180°
- Further illustrations: `https://medium.com/@2017csm1006/forward-and-backpropagation-in-convolutional-neural-network-4dfa96d7b37e`
  - Note that notation differs: error E there is cost C here, X there is z here, and F are the weights w here, and O is a here, and there is no bias

Training Variations

UNIVERSITÄT BIELEFELD

Hessian Technique

- As usual, let $\mathbf{w} = (w_1, w_2, ...)$ be NN parameters (weights in particular), and C a cost function.
- By *Taylor's theorem*, we can write

$$C(w + \delta w) = C(w) + \sum_j \frac{\partial C}{\partial w_j} \delta w_j + \frac{1}{2} \sum_{jk} \delta w_j H_{jk} \delta w_k + \text{... terms of higher order} \tag{15}$$

where H is the *Hessian matrix*, defined by

$$H_{jk} = \frac{\partial^2 C}{\partial w_j \partial w_k} \tag{16}$$

Hessian Technique

- Writing (15) as

$$C(w+\delta w) = C(w) + \nabla C \delta w + \frac{1}{2}\delta w^T H \delta w + \ldots \tag{17}$$

- and discarding all terms of order greater than 2, we obtain

$$C(w+\delta w) \approx C(w) + \nabla C \delta w + \frac{1}{2}\delta w^T H \delta w \tag{18}$$

- The right hand side of (18) can be minimized by choosing

$$\delta w = -H^{-1}\nabla C \tag{19}$$

Hessian Technique

Suggests following algorithm for updating weights:

1. Choose a starting point **w**
2. Update **w** to $\mathbf{w}' = w - \eta H^{-1} \nabla C$
   - H and ∇C are computed at **w**
3. Iterate—until appropriate criteria are met

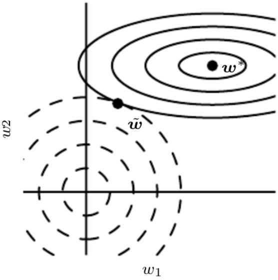
Hessian Technique

Advantages and Disadvantages

- The Hessian technique takes into account *how fast the gradient changes*
- Theoretical and empirical evidence: less iterations are needed
- *Issue:* Size of H is N^2 if N is the number of parameters
- ☞ Note that there could be $N = 10^7$ many parameters
- *Summary*:
  - Hessian technique often inapplicable because computations are too expensive
  - However, it provided inspiration for other techniques

Regularization Revisited

Motivation

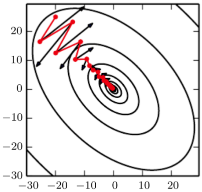


Adopted from `deeplearningbook.org`

- *Reminder*: L2 regularization shrinks weights along Hessian eigenvectors
- The ball then moves as being pulled by the origin $(0, 0)$ in the landscape induced by the eigenvectors of the Hessian

Momentum-based Gradient Descent

Motivation



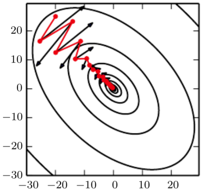
Black: gradients at each step, zig-zagging through the "valley"

Adopted from `deeplearningbook.org`

- *Motivation*: Going back and forth, without making progress, during (stochastic) gradient descent
- *Reasons*:
  - Poorly conditioned Hessian matrix because of "valleys" (see Figure)
  - Variance between batches
  - High curvature in general
  - Noisy gradients

Momentum-based Gradient Descent

Solution



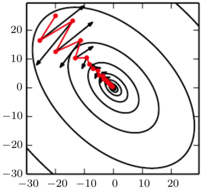
Red: averaged gradients, less zig-zagging

Adopted from `deeplearningbook.org`

- Keep track of earlier gradients and take the average

Momentum-based Gradient Descent

Solution



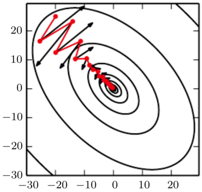
Red: averaged gradients, less zig-zagging

Adopted from `deeplearningbook.org`

- *Formally:* maintain *velocity* $\mathbf{v}$ in addition to parameters $\mathbf{w}$ themselves
- Let α be a *momentum hyperparameter*. In each iteration, update
  - velocity
    $$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_{\mathbf{w}}C \tag{20}$$
  - and parameters
    $$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v} \tag{21}$$

Momentum-based Gradient Descent

Solution



Red: averaged gradients, less zig-zagging

- In momentum based gradient descent, **w** is an exponentially decaying average over gradients
- Variant: *Nesterov's accelerated gradient technique*
- See also `http://neuralnetworksanddeeplearning.com/chap3.html` (Nielsen, chapter 3) for more details

Alternative Optimization

Further Reading

- *Alternative methods*:
  - *Conjugate gradient descent*
  - *BFGS (Broyden-Fletcher-Goldfarb-Shanno) method*
  - *L-BFGS (Limited-memory-BFGS) method*
- *Illustrations / Literature*:
  - Bengio's deep learning book: `http://www.deeplearningbook.org/contents/optimization.html`
  - "Efficient BackProp", Y. LeCun, L. Bottou, G. Orr, K.-R. Müller, 1998, see `http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf`
  - "On the importance of initialization and momentum in deep learning", I. Sutskever, J. Martens, G. Dahl and G. Hinton, 2012, `http://www.cs.toronto.edu/~hinton/absps/momentum.pdf`

Lecture5: Summary I

- Convolutional Neural Networks
  - `http://www.deeplearningbook.org/`, Chapter 9
  - `http://neuralnetworksanddeeplearning.com/`, "Deep Learning"
- Choosing hyperparameters
  - `http://www.deeplearningbook.org/`, Chapter 11 (selected parts)
  - `http://neuralnetworksanddeeplearning.com/`, "Weight initialization" and "How to choose a network's hyperparameters?"

LECTURE 5: SUMMARY II

- Convolutional backpropagation
  - For further illustrations, see `https://medium.com/@2017csm1006/forward-and-backpropagation-in-convolutional-neural-network-`
  - Note that there notation differs (error E there is cost C here, X there is z here, and F are the weights w here, and O is a here, and there is no bias)
- Training variations
  - `http://www.deeplearningbook.org/`, Chapter 8 (corresponding parts)
  - `http://neuralnetworksanddeeplearning.com/`, Chapter 3, "Variations on stochastic gradient descent"

OUTLOOK

- The vanishing gradient problem
  - `http://neuralnetworksanddeeplearning.com/`, Chapter 5
- Batch normalization
  - See `http://www.deeplearningbook.org/`, 8.7.1
  - See also `http://www.aifounded.com/machine-learning/deep-loss`, for example
- Recurrent neural networks
- Deep neural networks

Thanks for your attention