732A99/TDDE01/732A68 Machine Learning

Lecture 3d Block 1: Neural Networks II

Jose M. Peña
IDA, Linköping University, Sweden

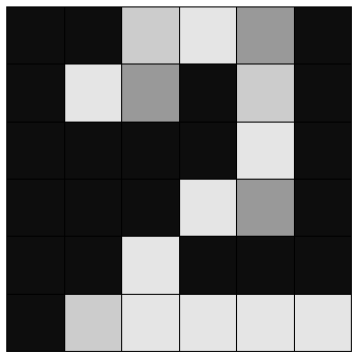
Contents and Literature

- Content
  - Convolutional Neural Networks
  - Regularization for Neural Networks
  - Strengths and Weaknesses of Neural Networks
  - Summary
- Literature
  - Lindholm, A., Wahlström, N., Lindsten, F. and Schön, T. B. *Machine Learning - A First Course for Engineers and Scientists*. Cambridge University Press, 2022. Chapter 6.3-6.4.
  - See also
    - `https://poloclub.github.io/cnn-explainer`

Convolutional Neural Networks (CNNs)

- CNNs are tailored for problems where the input data has a **grid-like structure**, e.g. images. Using ordinary NNs on the vectorized input data may miss the spatial patterns.

Image



Data representation

0.0	0.0	0.8	0.9	0.6	0.0
0.0	0.9	0.6	0.0	0.8	0.0
0.0	0.0	0.0	0.0	0.9	0.0
0.0	0.0	0.0	0.9	0.6	0.0
0.0	0.0	0.9	0.0	0.0	0.0
0.0	0.8	0.9	0.9	0.9	0.9

Input variables

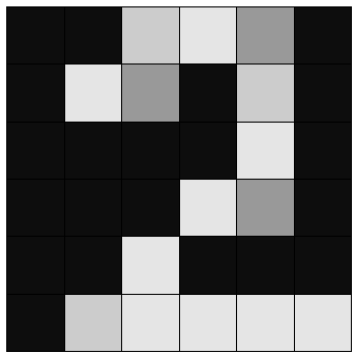
$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$

Figure 6.9: Data representation of a grayscale image with 6×6 pixels. Each pixel is represented with a number encoding the intensity of that pixel. These pixel values are stored in a matrix with the elements $x_{j,k}$.

Convolutional Neural Networks (CNNs)

- CNNs are tailored for problems where the input data has a **grid-like structure**, e.g. images. Using ordinary NNs on the vectorized input data may miss the spatial patterns.

Image



Data representation

0.0	0.0	0.8	0.9	0.6	0.0
0.0	0.9	0.6	0.0	0.8	0.0
0.0	0.0	0.0	0.0	0.9	0.0
0.0	0.0	0.0	0.9	0.6	0.0
0.0	0.0	0.9	0.0	0.0	0.0
0.0	0.8	0.9	0.9	0.9	0.9

Input variables

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$

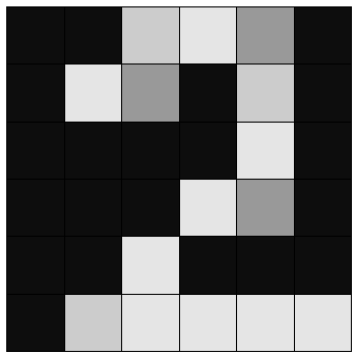
Figure 6.9: Data representation of a grayscale image with 6×6 pixels. Each pixel is represented with a number encoding the intensity of that pixel. These pixel values are stored in a matrix with the elements $x_{j,k}$.

- Ordinary NNs have dense layers: All the hidden units in a layer are connected to all the hidden units in the next layer, and each connection has a unique parameter.
- Instead, CNNs use **sparse interactions** and **parameter sharing**. This makes CNNs fairly invariant to translations of objects in the image. For instance, they will discover corners, edges, etc. regardless of where they are in the image.

Convolutional Neural Networks (CNNs)

- CNNs are tailored for problems where the input data has a **grid-like structure**, e.g. images. Using ordinary NNs on the vectorized input data may miss the spatial patterns.

Image



Data representation

0.0	0.0	0.8	0.9	0.6	0.0
0.0	0.9	0.6	0.0	0.8	0.0
0.0	0.0	0.0	0.0	0.9	0.0
0.0	0.0	0.0	0.9	0.6	0.0
0.0	0.0	0.9	0.0	0.0	0.0
0.0	0.8	0.9	0.9	0.9	0.9

Input variables

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$

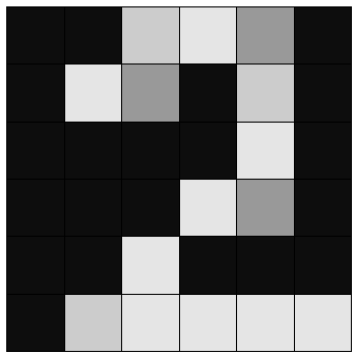
Figure 6.9: Data representation of a grayscale image with 6×6 pixels. Each pixel is represented with a number encoding the intensity of that pixel. These pixel values are stored in a matrix with the elements $x_{j,k}$.

- Ordinary NNs have dense layers: All the hidden units in a layer are connected to all the hidden units in the next layer, and each connection has a unique parameter.
- Instead, CNNs use **sparse interactions** and **parameter sharing**. This makes CNNs fairly invariant to translations of objects in the image. For instance, they will discover corners, edges, etc. regardless of where they are in the image.
- Moreover, CNNs use **stride** and **pooling** for summarizing information and strengthening the features in the image.

Convolutional Neural Networks (CNNs)

- CNNs are tailored for problems where the input data has a **grid-like structure**, e.g. images. Using ordinary NNs on the vectorized input data may miss the spatial patterns.

Image



Data representation

0.0	0.0	0.8	0.9	0.6	0.0
0.0	0.9	0.6	0.0	0.8	0.0
0.0	0.0	0.0	0.0	0.9	0.0
0.0	0.0	0.0	0.9	0.6	0.0
0.0	0.0	0.9	0.0	0.0	0.0
0.0	0.8	0.9	0.9	0.9	0.9

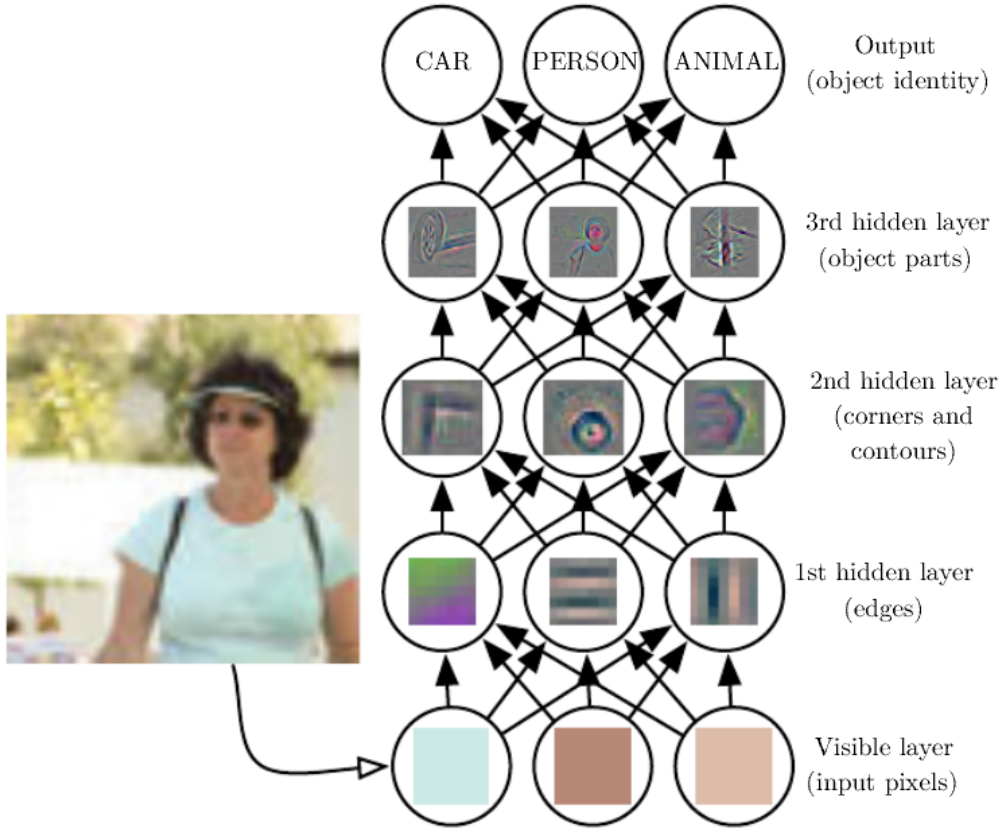
Input variables

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$

Figure 6.9: Data representation of a grayscale image with 6×6 pixels. Each pixel is represented with a number encoding the intensity of that pixel. These pixel values are stored in a matrix with the elements $x_{j,k}$.

- Ordinary NNs have dense layers: All the hidden units in a layer are connected to all the hidden units in the next layer, and each connection has a unique parameter.
- Instead, CNNs use **sparse interactions** and **parameter sharing**. This makes CNNs fairly invariant to translations of objects in the image. For instance, they will discover corners, edges, etc. regardless of where they are in the image.
- Moreover, CNNs use **stride** and **pooling** for summarizing information and strengthening the features in the image.
- Finally, CNNs use **multiple channels**, each of them sensitive to different features in the image.

Convolutional Neural Networks (CNNs)



CNNs: Sparse Interactions and Parameter Sharing

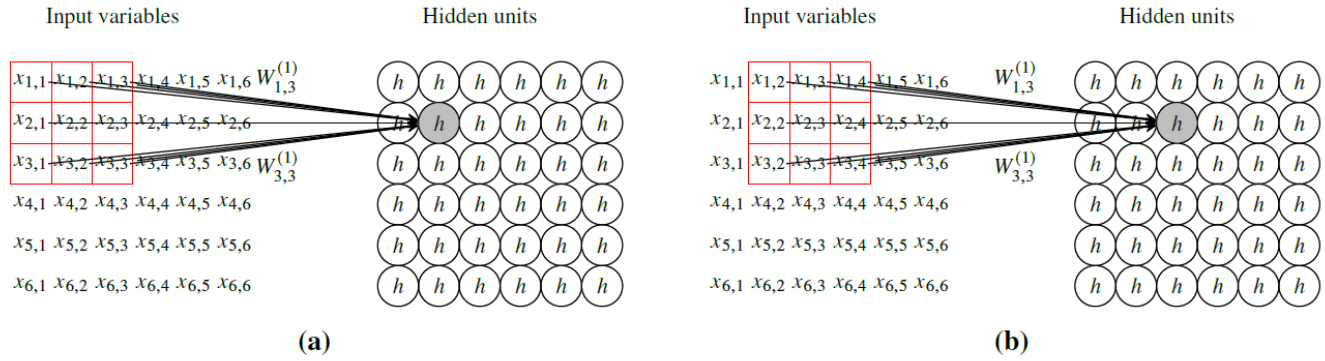


Figure 6.10: An illustration of the interactions in a convolutional layer: Each hidden unit (circle) is only dependent on the pixels in a small region of the image (red boxes), here of size 3×3 pixels. The location of the hidden unit corresponds to the location of the region in the image: if we move to a hidden unit one step to the right, the corresponding region in the image also moves one step to the right, compare Figure 6.10a and Figure 6.10b. Furthermore, the nine parameters $W^{(1)}_{1,1}, W^{(1)}_{1,2}, \ldots, W^{(1)}_{3,3}$ are the *same* for all hidden units in the layer.

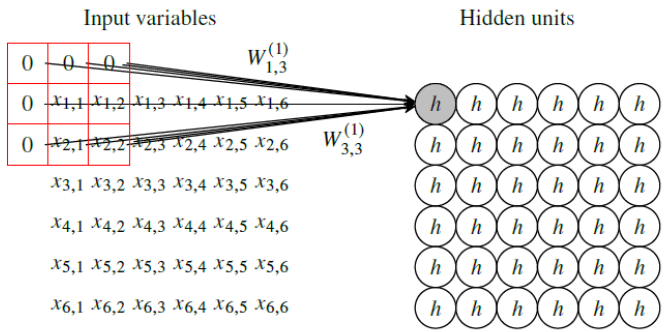


Figure 6.11: An illustration of zero-padding used when the region is partly outside the image. With zero-padding, the size of the image can be preserved in the following layer.

$$q_{ij} = h\left(\sum_{k=1}^{F}\sum_{l=1}^{F} x_{i+k-1,j+l-1} W_{k,l}\right), \tag{6.29}$$

where $x_{i,j}$ denotes the zero-padded input to the layer, q_{ij} the output of the layer, and $W_{k,l}$ the filter with F rows and F columns.

CNNs: Stride and Pooling

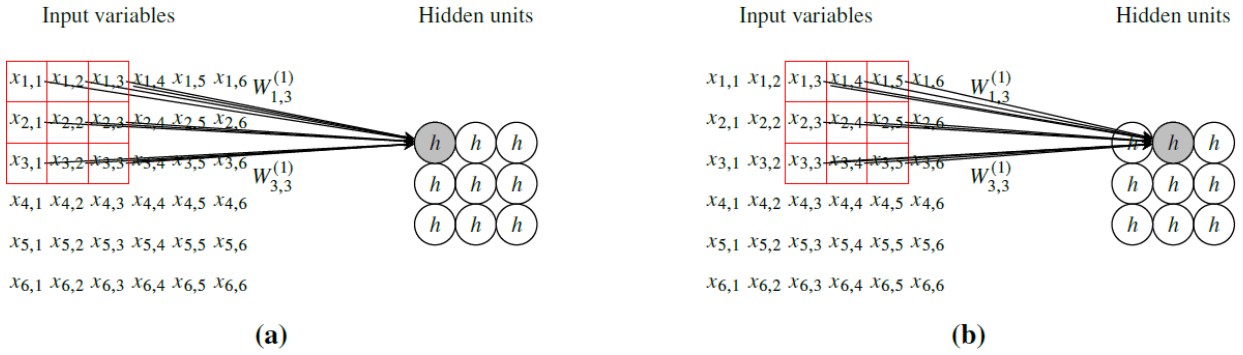


Figure 6.12: A convolutional layer with stride 2 and filter size 3×3.

$$q_{ij} = h\left(\sum_{k=1}^{F}\sum_{l=1}^{F} x_{s(i-1)+k,s(j-1)+l} W_{k,l}\right). \tag{6.30}$$

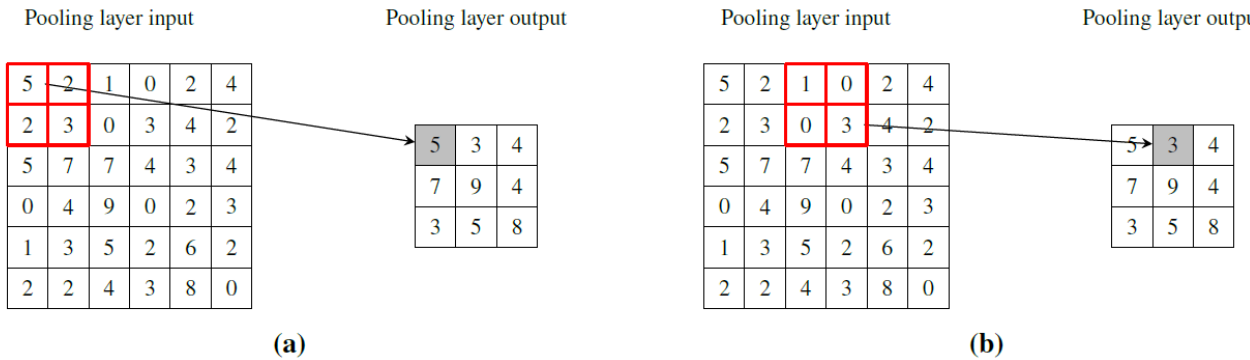


Figure 6.13: A max pooling layer with stride 2 and pooling filter size 2×2.

In average pooling,

$$\tilde{q}_{ij} = \frac{1}{F^2}\sum_{k=1}^{F}\sum_{l=1}^{F} q_{s(i-1)+k,s(j-1)+l}, \tag{6.31}$$

where $q_{i,j}$ is the input to the pooling layer, $\tilde{q}_{ij}$ is the output and, F is the pooling size and s is the stride used in the pooling layer.

CNNs: Multiple Channels

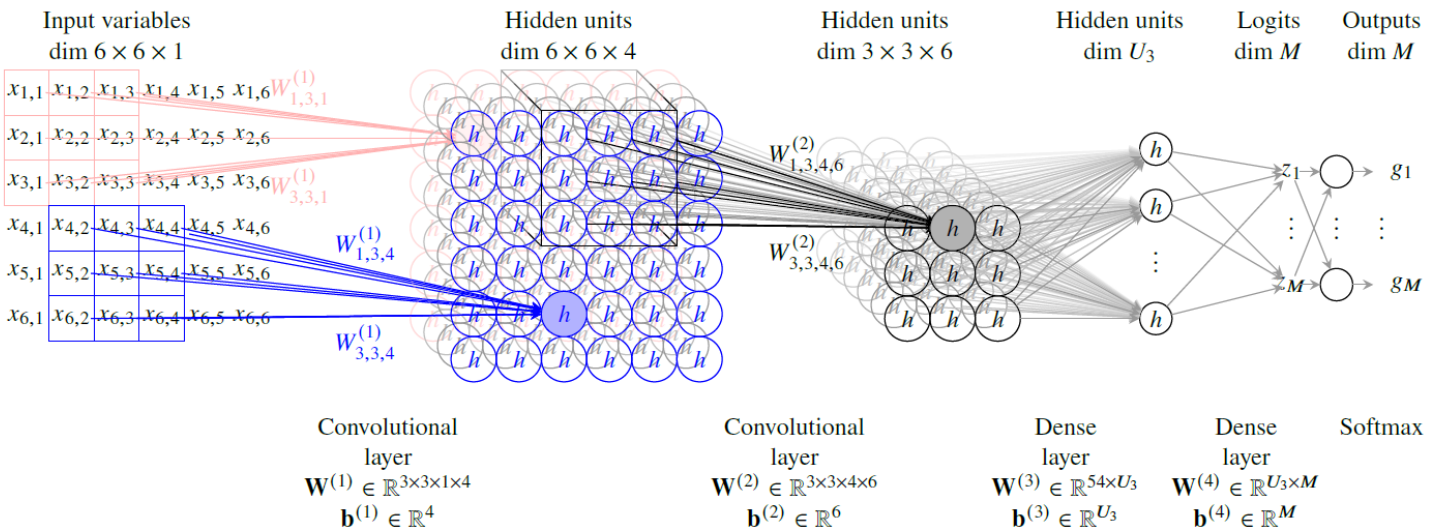


Figure 6.14: A full CNN architecture for classification of grayscale 6×6 images. In the first convolutional layer four filters each of size 3×3 produce a hidden layer with four channels. The first channel (in the back) is visualized in red and the forth channel (in the front) is visualized in blue. We use stride 1 which maintains the number of rows and columns. In the second convolutional layer, six filters of size $3 \times 3 \times 4$ and the stride 2 are used. They produce a hidden layer with 3 rows, 3 columns and 6 channels. After the two convolutional layers follows a dense layer where all $3 \cdot 3 \cdot 6 = 54$ hidden units in the second hidden layer are densely connected to all U_3 hidden units in the third layer, where all links have their unique parameters. We add an additional dense layer mapping down to the M logits. The network ends with a softmax function to provide predicted class probabilities as output. Note that the arrows corresponding to the offset parameters are not included here in order to make the figure less cluttered.

$$q^{(l)}_{ijn} = h\left(\sum_{k=1}^{F_l}\sum_{l=1}^{F_l}\sum_{m=1}^{U_{l-1}} q^{(l-1)}_{s_l(i-1)+k-1,s_l(j-1)+l,m} W^{(l)}_{k,l,m,n}\right), \tag{6.32}$$

where $q^{(l-1)}_{ijm}$ is the input to layer l, $q^{(l)}_{ijn}$ is the output from layer l, U_{l-1} is the number of input channels, F_l is the filter rows/columns, s_l is the stride, and $W^{(l)}_{k,l,m,n}$ is the weight tensor.

Example 6.3: Classification of hand-written digits - convolutional neural network

In the previous models explained in Example 6.1 and Example 6.2 we placed all the 28×28 pixels of each image into a long vector with 784 elements. With this action, we did not exploit the information that two neighboring pixels are more likely to be correlated that two pixels further apart. Instead, we now keep the matrix structure of the data and use a CNN with three convolutional layers and two dense layers. The settings for the layers are given in table below.

	Convolutional layers			Dense layers	
	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Number of filters/output channels	4	8	12	-	-
Filter rows and columns	(5×5)	(5×5)	(4×4)	-	-
Stride	1	2	2	-	-
Number of hidden units	3136	1568	588	200	10
Number of parameters (including offset vector)	104	808	1548	117800	2010

In a high-dimensional parameter space the number of saddle points in the cost function are frequent. Since the gradients are zero also at these saddle points, stochastic gradient descent might get stuck there. Therefore, we train this model using an extension of stochastic gradient descent called Adam (short for adaptive moment estimation). Adam is using running averages on the gradients and its second order moments and can pass these saddle points more easily. For the Adam optimizer we use a learning rate of $\gamma = 0.002$. In Figure 6.15, the cost function and the misclassification rate on the current training mini-batch and the validation data is displayed. The shaded red line is the performance on the validation data for the two-layer network that was presented in Example 6.2.

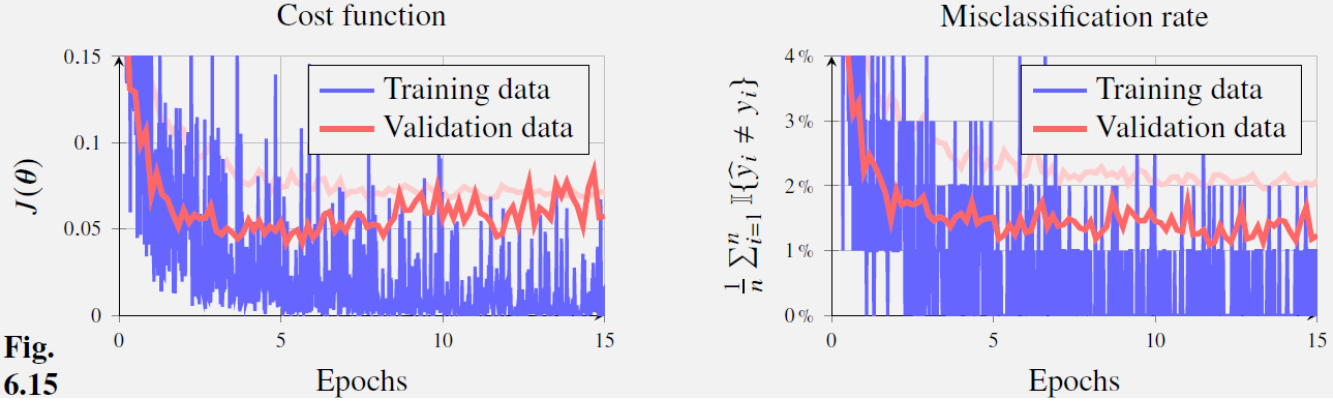


Fig. 6.15

In comparison to the result on the dense two-layer network we can see an improvement going from 2% down to just over 1% misclassification on the validation data. From Figure 6.15 we can also see that the performance on validation data does not quite settle but is fluctuating in the span 1%–1.5%. As explained in Section 5.5, this is due to the randomness introduced by stochastic gradient descent itself. To circumvent this effect we use an decaying learning rate. We use the scheme suggested in (5.39) with $\gamma_{\max} = 0.003$ and $\gamma_{\min} = 0.0001$ and $\tau = 2000$ (Figure 6.16).

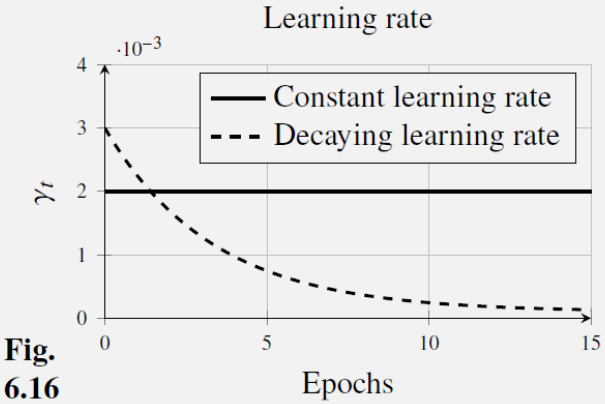
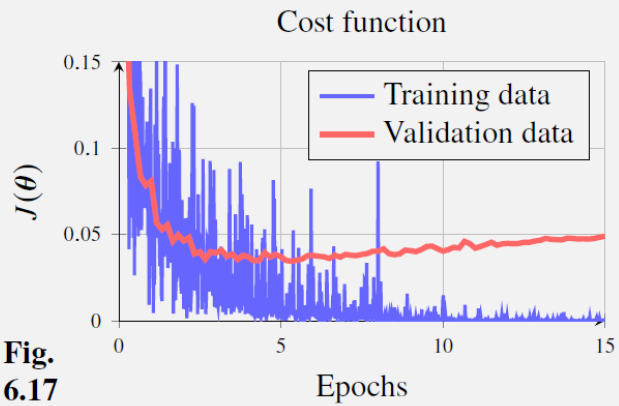


Fig. 6.16



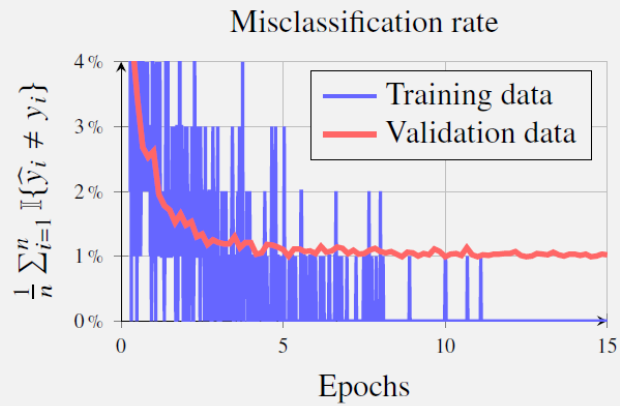


Fig. 6.17

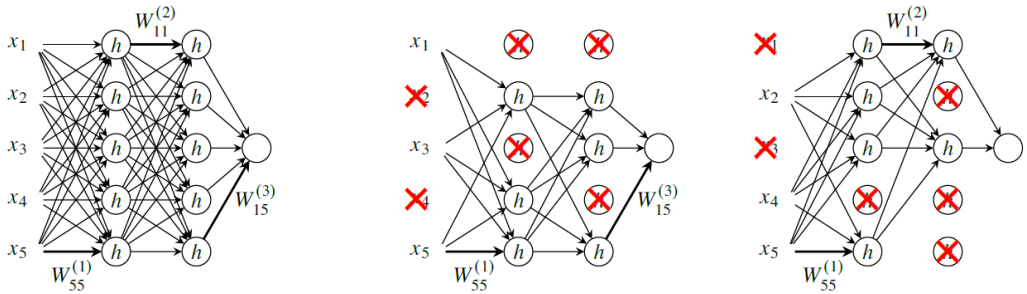
After employing the adaptive learning rate in Figure 6.17, the misclassification rate on validation data settles around 1% and not only sometimes bouncing down to 1% before we applied an decaying learning rate. However, we can do more. In the last epochs, we get almost all data points correct in the current mini-batch. In addition, looking at the plot for the cost function evaluated for the validation data, it starts increasing after 5 epochs. Hence, we see signs of overfitting as we did also in the end of Example 6.2. To circumvent this overfitting, we can add regularization. One popular regularization method for neural networks is dropout, which is explained in the following section.

Regularization for NNs

- Dropout is a technique **to reduce the risk of overfitting** in NNs, by reducing the variance of a NN's predictions without increasing the bias.

Regularization for NNs

- Dropout is a technique **to reduce the risk of overfitting** in NNs, by reducing the variance of a NN's predictions without increasing the bias.
- Dropout at training time: In each gradient descent step, use a different minibatch and sub-network. The sub-network is obtained by dropping each input and hidden unit with probability $1 - r$.



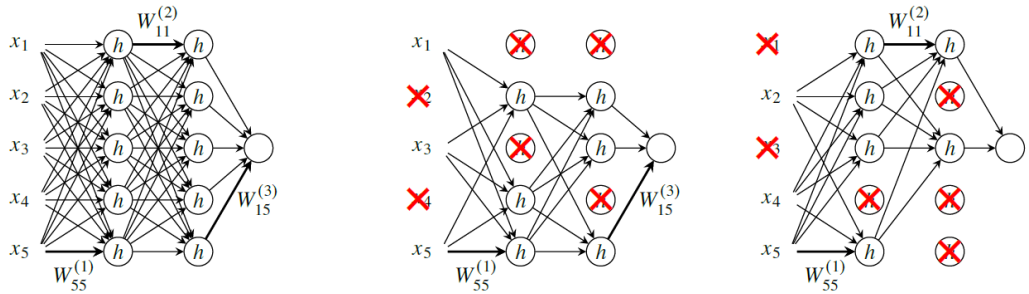
(a) A standard neural network

(b) Two sub-networks

Figure 6.18: A neural network with two hidden layers (a), and two sub-networks with dropped units (b). The collection of units that have been dropped are independent between the two sub-networks.

Regularization for NNs

- Dropout is a technique **to reduce the risk of overfitting** in NNs, by reducing the variance of a NN's predictions without increasing the bias.
- Dropout at training time: In each gradient descent step, use a different minibatch and sub-network. The sub-network is obtained by dropping each input and hidden unit with probability $1 - r$.



(a) A standard neural network **(b)** Two sub-networks

Figure 6.18: A neural network with two hidden layers (a), and two sub-networks with dropped units (b). The collection of units that have been dropped are independent between the two sub-networks.

- Dropout at prediction time:

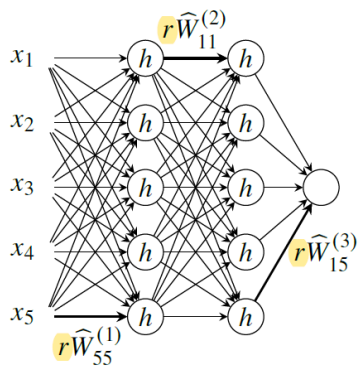


Figure 6.19: The network used for prediction after being trained with dropout. All units and links are present (no dropout) but the weights going out from a certain unit are multiplied with the probability of that unit being included during training. This is to compensate for the fact that some of them were dropped during training. Here all units have been kept with probability r during training (and consequently dropped with probability $1 - r$).

Regularization for NNs

Example 6.4: Classification of hand-written digits - regularizing with dropout

We return to the last model in Example 6.3 where we used a CNN trained with an adaptive learning rate. In the results, we could see clear indications of overfitting. To reduce this overfitting we employ dropout during the training procedure. We use dropout in the final hidden layer and keep only $r = 75\%$ of the 200 hidden units in that layer at each iteration. In Figure 6.20 the result for this regularized model is presented. In shaded red we also present the performance on validation data for the non-regularized version which was already presented in the end of Example 6.3.

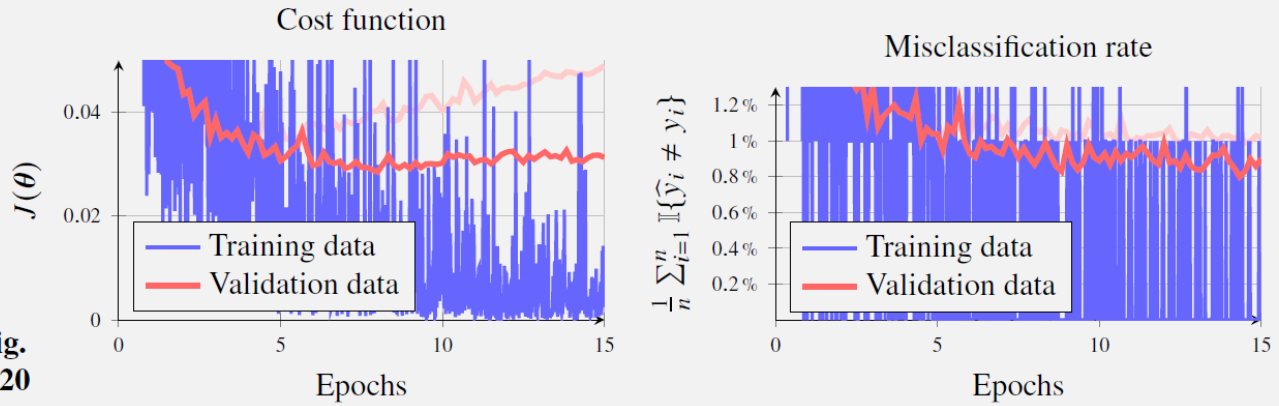


Fig. 6.20

In contrast to the last model of Example 6.3, the cost function evaluated for the validation data is (almost) no longer increasing and we also reduce the misclassification rate by an additional 0.1% to 0.2%.

Regularization for NNs

Example 6.4: Classification of hand-written digits - regularizing with dropout

We return to the last model in Example 6.3 where we used a CNN trained with an adaptive learning rate. In the results, we could see clear indications of overfitting. To reduce this overfitting we employ dropout during the training procedure. We use dropout in the final hidden layer and keep only $r = 75\%$ of the 200 hidden units in that layer at each iteration. In Figure 6.20 the result for this regularized model is presented. In shaded red we also present the performance on validation data for the non-regularized version which was already presented in the end of Example 6.3.

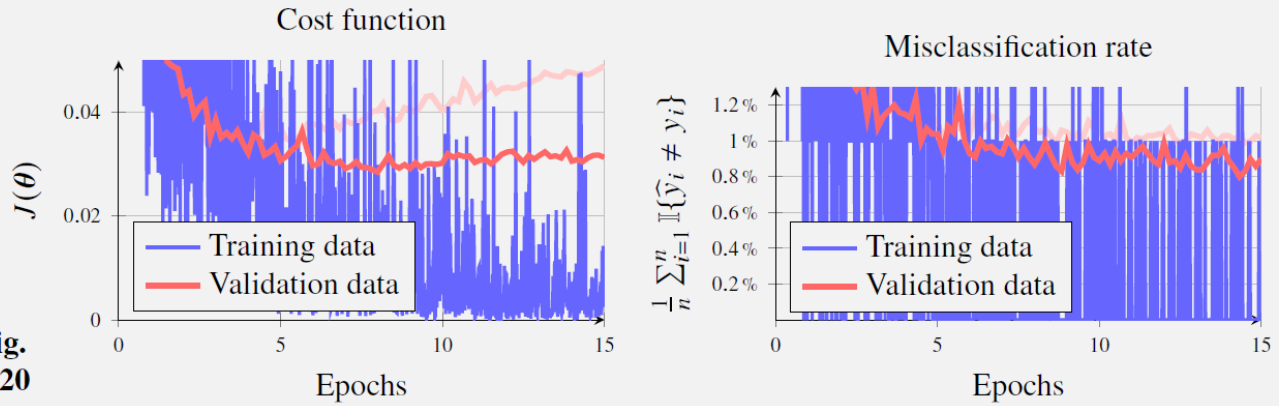


Fig. 6.20

In contrast to the last model of Example 6.3, the cost function evaluated for the validation data is (almost) no longer increasing and we also reduce the misclassification rate by an additional 0.1% to 0.2%.

- Dropout may be the most popular regularization method for NNs. Other methods include L^1 or L^2 regularization, early stopping, and sparse representations similar to CNNs.

Regularization for NNs

Example 6.4: Classification of hand-written digits - regularizing with dropout

We return to the last model in Example 6.3 where we used a CNN trained with an adaptive learning rate. In the results, we could see clear indications of overfitting. To reduce this overfitting we employ dropout during the training procedure. We use dropout in the final hidden layer and keep only $r = 75\%$ of the 200 hidden units in that layer at each iteration. In Figure 6.20 the result for this regularized model is presented. In shaded red we also present the performance on validation data for the non-regularized version which was already presented in the end of Example 6.3.

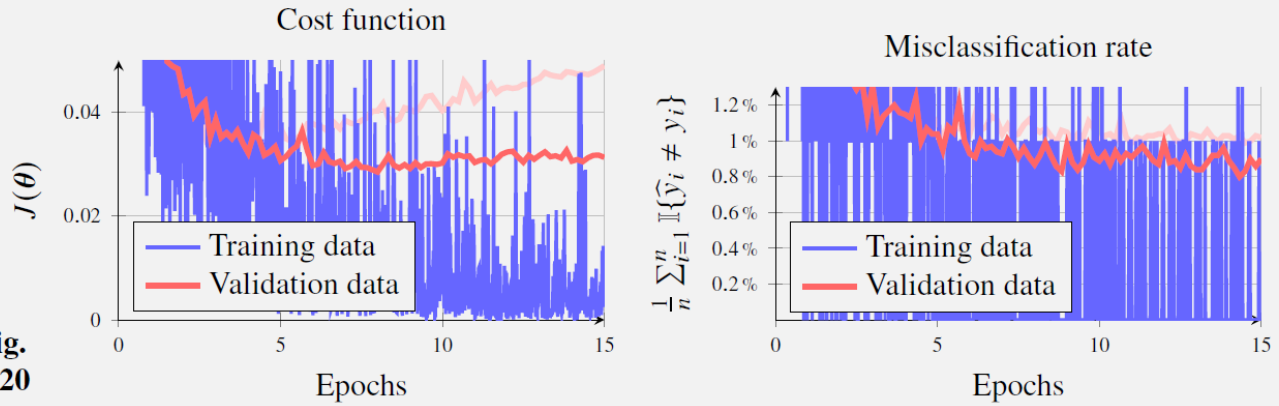


Fig. 6.20

In contrast to the last model of Example 6.3, the cost function evaluated for the validation data is (almost) no longer increasing and we also reduce the misclassification rate by an additional 0.1% to 0.2%.

- Dropout may be the most popular regularization method for NNs. Other methods include L^1 or L^2 regularization, early stopping, and sparse representations similar to CNNs.
- Only for 732A99: Dropout resembles bagging, where several predictors trained on bootstrapped samples of the original data are averaged with the same purpose. Bagging is simply too computationally demanding.

Strengths and Weaknesses of NNs

Theorem (Universal approximation theorem)

For every continuous function $f : [a, b]^D \to \mathbb{R}$ *and for every* $\epsilon > 0$*, there exists a NN with one hidden layer such that*

$$\sup_{\mathbf{x} \in [a,b]^D} |f(\mathbf{x}) - y(\mathbf{x})| < \epsilon$$

Strengths and Weaknesses of NNs

Theorem (Universal approximation theorem)

For every continuous function $f : [a, b]^D \to \mathbb{R}$ *and for every* $\epsilon > 0$*, there exists a NN with one hidden layer such that*

$$\sup_{\mathbf{x} \in [a,b]^D} |f(\mathbf{x}) - y(\mathbf{x})| < \epsilon$$

Theorem (Universal classification theorem)

Let $\mathcal{C}^{(k)}$ *contain all classifiers defined by NNs of one hidden layer with* k *hidden units and the sigmoid activation function. Then, for any distribution* $p(\mathbf{x}, t)$*,*

$$\lim_{k \to \infty} \inf_{y \in \mathcal{C}^{(k)}} L(y(\mathbf{x})) - L(p(t|\mathbf{x})) = 0$$

where $L(\cdot)$ *is the 0/1 loss function.*

Strengths and Weaknesses of NNs

Theorem (Universal approximation theorem)

For every continuous function $f : [a, b]^D \to \mathbb{R}$ *and for every* $\epsilon > 0$*, there exists a NN with one hidden layer such that*

$$\sup_{\mathbf{x} \in [a,b]^D} |f(\mathbf{x}) - y(\mathbf{x})| < \epsilon$$

Theorem (Universal classification theorem)

Let $\mathcal{C}^{(k)}$ *contain all classifiers defined by NNs of one hidden layer with* k *hidden units and the sigmoid activation function. Then, for any distribution* $p(\mathbf{x}, t)$*,*

$$\lim_{k \to \infty} \inf_{y \in \mathcal{C}^{(k)}} L(y(\mathbf{x})) - L(p(t|\mathbf{x})) = 0$$

where $L(\cdot)$ *is the 0/1 loss function.*

- How many hidden units have such NNs?

Strengths and Weaknesses of NNs

Theorem (Universal approximation theorem)

For every continuous function $f : [a, b]^D \to \mathbb{R}$ and for every $\epsilon > 0$, there exists a NN with one hidden layer such that

$$\sup_{\mathbf{x} \in [a,b]^D} |f(\mathbf{x}) - y(\mathbf{x})| < \epsilon$$

Theorem (Universal classification theorem)

Let $\mathcal{C}^{(k)}$ contain all classifiers defined by NNs of one hidden layer with k hidden units and the sigmoid activation function. Then, for any distribution $p(\mathbf{x}, t)$,

$$\lim_{k \to \infty} \inf_{y \in \mathcal{C}^{(k)}} L(y(\mathbf{x})) - L(p(t|\mathbf{x})) = 0$$

where $L(\cdot)$ is the 0/1 loss function.

- How many hidden units have such NNs?
- How much data do we need to train such NNs (and avoid overfitting) via the backpropagation algorithm?

Strengths and Weaknesses of NNs

Theorem (Universal approximation theorem)

For every continuous function $f : [a, b]^D \to \mathbb{R}$ and for every $\epsilon > 0$, there exists a NN with one hidden layer such that

$$\sup_{\mathbf{x} \in [a,b]^D} |f(\mathbf{x}) - y(\mathbf{x})| < \epsilon$$

Theorem (Universal classification theorem)

Let $\mathcal{C}^{(k)}$ contain all classifiers defined by NNs of one hidden layer with k hidden units and the sigmoid activation function. Then, for any distribution $p(\mathbf{x}, t)$,

$$\lim_{k \to \infty} \inf_{y \in \mathcal{C}^{(k)}} L(y(\mathbf{x})) - L(p(t|\mathbf{x})) = 0$$

where $L(\cdot)$ is the 0/1 loss function.

- How many hidden units have such NNs?
- How much data do we need to train such NNs (and avoid overfitting) via the backpropagation algorithm?
- How fast does the backpropagation algorithm converge to such NNs? Assuming that it does not get trapped in a local minimum...

Strengths and Weaknesses of NNs

Theorem (Universal approximation theorem)

For every continuous function $f : [a, b]^D \to \mathbb{R}$ and for every $\epsilon > 0$, there exists a NN with one hidden layer such that

$$\sup_{\mathbf{x} \in [a,b]^D} |f(\mathbf{x}) - y(\mathbf{x})| < \epsilon$$

Theorem (Universal classification theorem)

Let $\mathcal{C}^{(k)}$ contain all classifiers defined by NNs of one hidden layer with k hidden units and the sigmoid activation function. Then, for any distribution $p(\mathbf{x}, t)$,

$$\lim_{k \to \infty} \inf_{y \in \mathcal{C}^{(k)}} L(y(\mathbf{x})) - L(p(t|\mathbf{x})) = 0$$

where $L(\cdot)$ is the 0/1 loss function.

- How many hidden units have such NNs?
- How much data do we need to train such NNs (and avoid overfitting) via the backpropagation algorithm?
- How fast does the backpropagation algorithm converge to such NNs? Assuming that it does not get trapped in a local minimum...
- The answer to the last two questions depends on the first: More hidden units implies more training time and larger risk of overfitting.

Strengths and Weaknesses of NNs

Theorem (Universal approximation theorem)

For every continuous function $f : [a, b]^D \to \mathbb{R}$ and for every $\epsilon > 0$, there exists a NN with one hidden layer such that

$$\sup_{\mathbf{x} \in [a,b]^D} |f(\mathbf{x}) - y(\mathbf{x})| < \epsilon$$

Theorem (Universal classification theorem)

Let $\mathcal{C}^{(k)}$ contain all classifiers defined by NNs of one hidden layer with k hidden units and the sigmoid activation function. Then, for any distribution $p(\mathbf{x}, t)$,

$$\lim_{k \to \infty} \inf_{y \in \mathcal{C}^{(k)}} L(y(\mathbf{x})) - L(p(t|\mathbf{x})) = 0$$

where $L(\cdot)$ is the 0/1 loss function.

- How many hidden units have such NNs?
- How much data do we need to train such NNs (and avoid overfitting) via the backpropagation algorithm?
- How fast does the backpropagation algorithm converge to such NNs? Assuming that it does not get trapped in a local minimum...
- The answer to the last two questions depends on the first: More hidden units implies more training time and larger risk of overfitting.

Theorem (No free lunch theorem)

For any method, good performance on some problems comes at the expense of bad performance on some others.

Summary

- Convolutional Neural Networks
- Regularization for Neural Networks
- Strengths and Weaknesses of Neural Networks