Part III

Image analysis with neural networks

NEURAL NETWORKS are very useful for a range of image analysis tasks including segmentation, detection, classification, etc. Neural networks are often easy to adapt to a specific problem and they allow approximating an unknown function that, based on some input $\mathbf{x}$, can predict the output $\mathbf{y}$ even without *a priori* knowing the relation between $\mathbf{x}$ and $\mathbf{y}$. This is done by parameterizing the function and learning its parameters using a training set, i.e. a set of many pairs of input values and corresponding predictions. In image analysis problems, the input will typically be an image or a part of an image, and the output is a scalar vector or an image.

A range of high-performance libraries for neural networks exists that are very well suited for solving many problems also in image analysis. The aim here is, however, to give an understanding of the basic elements of neural networks and get experience with their functionality. This will be done by implementing a feed-forward neural network, a Multilayer Perceptron (MLP). The first task is to separate simple point sets. This is not an image analysis task, but it is chosen as a simple and easy-to-visualize task that can help verify that the implementation is correct. Furthermore, it will give you some experience with various model parameters. This implementation will later be applied to image classification and image segmentation.

8 Feed forward neural network

This exercise is based on the description in the Deep Learning book[1] which covers both the fundamentals and the details of the deep learning methods. Chapter 5 in the book Pattern Recognition and Machine Learning[2] also gives a good introduction to neural networks, and our notation resembles the notation used in the book by Bishop.

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016

[2] CM Bishop. Pattern recognition and machine learning., 2006

8.1 Concept of neural network

A neural network is often drawn as a directed graph as shown in Figure 8.1. The input layer is shown on the left, hidden layers are in the middle, and the output layer is to the right. Here, we will give a brief introduction to a simple feed-forward network. You will later implement such a network.

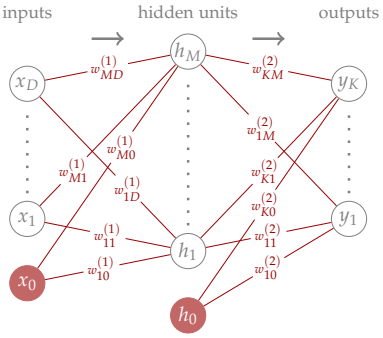

Figure 8.1: Example of a neural network with an input layer, one hidden layer, and an output layer. This is termed a one-layer network since it has one hidden layer. Typically, there will be many hidden layers.

Conceptually we want to construct a function f that takes a vector $\mathbf{x}$ as input and predicts a vector $\mathbf{y}$

$$f(\mathbf{x}) = \mathbf{y}.$$

In image analysis, the input $\mathbf{x}$ will often be an image reshaped into a vector (elements of the vector are all pixel values). The output depends on the problem to be solved. If we deal with a classification problem, the output will typically be a vector of class probabilities. E.g. if there are K classes, $\mathbf{y}$ will be a K-dimensional vector, where each element in the vector is the probability of belonging to one of the K classes.

One can also construct a neural network for other tasks, for example image segmentation. When segmenting an image consisting of D pixels into L labels, then the output will be an $D \times L$ matrix where each element is a probability of the i-th pixel belonging to the j-th label where $i = 1, ..., D$ and $j = 1, ..., L$.

There are steps in deep learning that require design choices. This includes choices such as the number of layers, the number of nodes in these layers, and decisions regarding regularization parameters. Such settings are referred to as hyper-parameters of the model (as opposed

to the edge weights that are the model parameters and will be updated during training).

We will start by explaining the *forward model* of the feed-forward neural network. Through the forward pass, we get the model prediction, Once the network is trained, the predictions will solve our image analysis problem. But to construct a model that gives us the desired results, we must first tune the parameters of the model. This is done by changing the edge weights such that the model can correctly predict the values of a training set. When tuning the model parameters we use the *backpropagation algorithm*, which will be explained after the forward model.

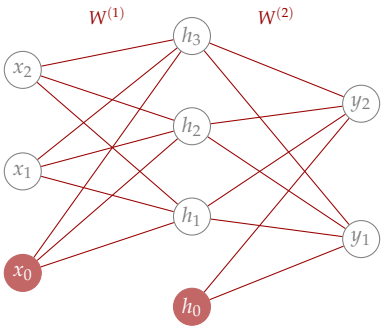

Figure 8.2: Simple three layer neural network.

8.2 Forward model

For simplicity, we start by describing the network shown in Figure 8.2. This network will solve a classification problem. The network takes a two-dimensional input vector (x_1, x_2) and outputs a probability for two classes. The network contains an input layer of three nodes (also called neurons), where two take the values of the input x_1 and x_2 (independent variables), and one node, called the bias node, is set to $x_0 = 1$. The hidden layer contains four nodes including three nodes connected to the input layer (h_1, h_2, h_3) and the bias node $h_0 = 1$. The output layer contains the two predicted values (y_1, y_2) (dependent variables). The weights of the edges connecting the nodes are termed $w_{ij}^{(l)}$ for the edge connecting node j from layer $l-1$ with node i in layer l.

The values of the nodes in the hidden layers are computed by first computing a weighted linear combination z_i of the node values and the edge weights followed by a non-linear activation function. Here we use the max function $a(z_i) = \max\{z_i, 0\}$ to obtain h_i. In deep learning, this function is called the rectified linear units function (ReLU). We have

$$z_i = \sum_{d=0}^{D} w_{id}^{(1)} x_d \,, \quad D = 2, \quad i = 1, \ldots, 3, \tag{8.1}$$

$$h_i = a(z_i) = \max\{0, z_i\} \,, \tag{8.2}$$

$$\hat{y}_j = \sum_{m=0}^{M} w_{jm}^{(2)} h_m, \quad M = 3, \quad j = 1, 2 \,. \tag{8.3}$$

Since the output of the network should be used for classification, we want to interpret the output values as probabilities, i.e. the element y_j should be seen as the probability of the input belonging to the j-th class. Therefore, we must transform the values $\hat{y}_j$ into positive values summing to 1. For this, we use the softmax function

$$y_j = \frac{\exp \hat{y}_j}{\sum_{k=1}^{K} \exp \hat{y}_k} \,. \tag{8.4}$$

When solving a classification problem, each input will be classified as belonging to the output class j^* with the highest value of all y_j.

Note that the output of the network is defined by the set of network weights $w_{ij}^{(l)}$.

8.3 *Backpropagation*

We now want to train the network, i.e. adjust the weights of the network such that it predicts the desired output. For this, we define a loss function, which is a measure of how different is the current prediction from the desired prediction. To measure the loss we need some data for which we know the desired values, also called the target values.

During training, we minimize the loss over a training set of inputs and the associated target values. In training, we use the backpropagation algorithm to compute the gradient of the loss function concerning each weight. Then we use an iterative optimization method called stochastic gradient descent. Just as in gradient descent the loss is minimized by taking steps proportional to the negative gradient.

For a classification problem, we use the cross entropy loss function computed from the predicted value $\mathbf{y}$ and the target $\mathbf{t}$ as

$$L = -\sum_{k=1}^{K} t_k \ln y_k \,, \tag{8.5}$$

where t_k is the target value of the prediction where

$$t_k = \begin{cases} 1 & \text{if class label is } k \\ 0 & \text{otherwise} \end{cases} \,. \tag{8.6}$$

Since t_k has only one non-zero term, there is only one non-zero term in the summation from Eq. (8.5). So, the loss takes the value $L = -\ln y_{k^*}$, where k^* is the target class. This is a positive value unless $\mathbf{y} = \mathbf{t}$, in which case the loss is zero.

When training the network we use the predictions and targets for all data in our training set. However, when updating weights we do not consider the loss function for all inputs and targets at once. Instead, we consider one input (or a smaller sample of inputs called a minibatch) in a random order (therefore the term stochastic in the name of the optimization). One cycle through the full training dataset is called one epoch.

In each iteration of the stochastic gradient descent, we evaluate partial derivatives for a certain (fixed) input to determine how the change in each $w_{ij}^{(l)}$ affects L. Then we use an update

$$w_{ij}^{(l)\text{new}} = w_{ij}^{(l)} - \eta \frac{\partial L}{\partial w_{ij}^{(l)}} \,,$$

where η is a user-chosen learning rate.

In the derivation below we explain the computation of the partial derivatives using backpropagation. Change in each $w_{ij}^{(l)}$ contributes to the change in L only through $z_i^{(l)}$ and using the chain rule the derivative may be separated into two elements

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \overbrace{\frac{\partial L}{\partial z_i^{(l)}}}^{\text{difficult}} \overbrace{\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}}^{\text{easy}}.$$

Since $z_i^{(l)}$ is a linear function of $w_{ij}^{(l)}$ the second (easy) partial derivative evaluates to $h_j^{(l-1)}$ (or, in the case of the first layer, input values x_j). So, when implementing backpropagation, the node values need to be stored during the forward pass.

The first (more difficult) partial derivative needs to be evaluated for each $z_i^{(l)}$. We denote these values by $\delta_i^{(l)} = \frac{\partial L}{\partial z_i^{(l)}}$, such that we have

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} h_j^{(l-1)}. \tag{8.7}$$

Notice here that the update for weight $w_{ij}^{(l)}$ is a product of two values, the first value depends only on the *to*-node, and the second value depends only on the *from*-node.

We still need to evaluate $\delta_i^{(l)}$, i.e. establish how a change of $z_i^{(l)}$ affects L. This depends only on what happens in the layers further down the pipeline, and on the choice of the non-linear activation used on $z_i^{(l)}$. We distinguish between the last layer (where we use the softmax function) and the internal layers.

For the last layer (denoted l^*, check note[3]) we express L as a function of $z_k^{(l^*)}$

$$\begin{aligned} L &= -\sum_k t_k \ln \frac{\exp z_k^{(l^*)}}{\sum_j \exp z_j^{(l^*)}} = \overset{\text{using the properties of ln and the distributive rule}}{} \\ &= -\sum_k t_k z_k^{(l^*)} + \overbrace{\sum_k t_k}^{=1} \overbrace{\ln \sum_j \exp z_j^{(l^*)}}^{\text{equal for all } k}. \end{aligned}$$

The derivative of L with respect to $z_i^{(l^*)}$ is therefore

$$\delta_i^{(l^*)} = -t_i + \frac{1}{\sum_j \exp z_k^{(l^*)}} \exp z_i^{(l^*)} = y_i - t_i. \tag{8.8}$$

Now consider the internal layers. The change in $z_i^{(l)}$ may change all

[3] When introducing the three-layer network in Figure 8.2, we have used the term $\hat{y}_k$ for the non-activated values in the last layer to distinguish them from the values in the layer before. However, the computation of non-activated values is the same for all layers, and the difference is only in the activation function. So, for a general situation with many layers l, we will use the same notation for the non-activated values. With the last layer being l^*, the non-activated values in the last layer are now $z_k^{(l^*)}$.

z_k^{l+1}, and any of these changes may affect L. The chain rule gives

$$\frac{\partial L}{\partial z_i^{(l)}} = \sum_k \underbrace{\frac{\partial L}{\partial z_k^{(l+1)}}}_{\text{we have}} \underbrace{\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}}_{\text{we need}}.$$

The first set of derivatives are $\delta_k^{(l+1)}$ for the layer further down the pipeline. We already evaluated those for the last layer in (8.8), and this is why we compute the update backward through the network. The only remaining is to determine how the change of $z_i^{(l)}$ affects $z_k^{(l+1)}$. From definition (8.1) we see that $z_k^{(l+1)}$ is a linear function of $a\left(z_i^{(l)}\right)$ which gives

$$\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = w_{ki}^{(l+1)} a'\left(z_i^{(l)}\right),$$

where a' denotes the derivative of the activation function, which for ReLU function takes a value zero for arguments smaller than zero, and one otherwise. This is easy to determine by assessing whether $h_i^{(l)}$ is zero or larger. The final expression for internal layers is

$$\delta_i^{(l)} = a'\left(z_i^{(l)}\right) \sum_k w_{ki}^{(l+1)} \delta_k^{(l+1)}. \tag{8.9}$$

8.4 Implementation

You are now ready to implement a neural network with a forward and a backward pass as described above. Below, we give a few practical suggestions, and we outline the steps you should take when implementing the network.

8.4.1 2D clustering data

You will need some test data for running your method. For this, we have provided a file `make_data.py`. This code will create point sets similar to the ones shown in Figure 8.3. In this experiment, you will visualize the output of the network on the regular grid, so you can use all of the generated points for training. This first experiment is to ensure that your implementation of the neural network is correct.

Standardize data To ensure the numerical stability of the MLP, it is a good idea to standardize the input data to have zero mean and standard deviation of one in each dimension. Store the mean values and standard deviations of the original points, such that you can transform new input points using these values.

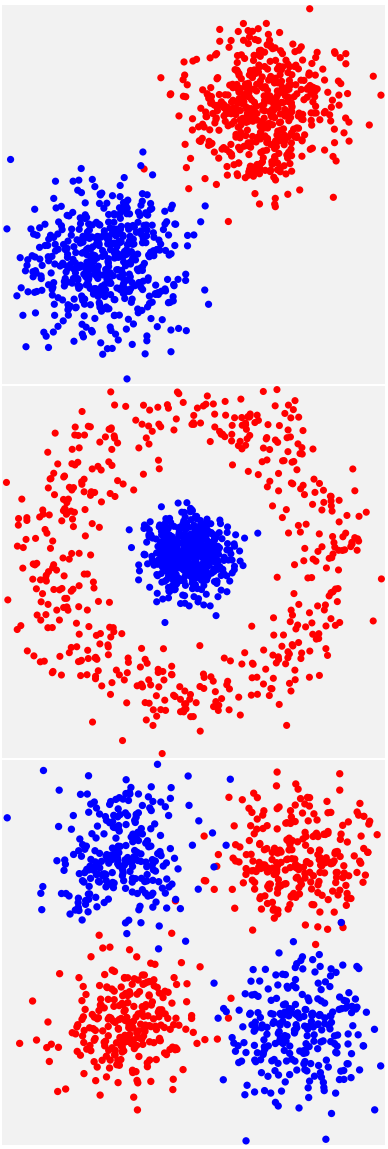

Figure 8.3: Scatter plots of point sets to test neural network

8.4.2 *Simple three layer network*

You should start with a three-layer network shown in Figure 8.2. This network contains a single hidden layer, it takes a two-dimensional input and classifies the input to a two-dimensional output. The hidden layer has four neurons where three are connected to the input layer and one is a bias neuron. It is a good idea to start making a hand drawing of the network you should implement with the nodes, edges, and notation for the parts of the network.

Storing weights as arrays Edge weights are best stored as 2D arrays that allow computing node values between layers using matrix-vector multiplications. We suggest that you store all weights for one layer in one array (i.e. you don't need a separate array for bias weights) but remember that this is an implementation choice.

In the steps suggested below, we keep bias weights in the *last row* of weight matrices, while in the drawings and notation bias weights are indexed with w_{i0}. So, deciding to store weights in the first (zeroth) column would be a choice that is perhaps closer to mathematical notation.

For the simple three-layer network, the weight arrays $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ will be of dimension 3-by-3 for mapping from layer 1 to 2 (input to hidden layers without bias) and 4-by-2 for mapping from layer 2 to 3 (hidden layer to outputs).

Keep good track of the array sizes! Figure 8.4 shows sizes of arrays for the simple three-layer network.

Initializing the weights You must initialize the weight arrays with random numbers that are scaled adequately. Initializing with zeros is not a good idea, as it introduces no asymmetry between neurons. If the weights are too large or too small, you might risk numerical instability where your network will not converge. The current recommendation in the case of neural networks with ReLU neurons is to initialize with normally distributed random numbers scaled with a factor $\sqrt{\frac{2}{n}}$, where n is the number of inputs to the neuron. I.e., use $n = 3$ when initializing $\mathbf{W}^{(1)}$ and $n = 4$ for $\mathbf{W}^{(2)}$.

Forward model The forward model includes a mapping from the input $\mathbf{x}$ to the hidden nodes with values $\mathbf{z}$. This can be computed as a matrix-vector multiplication

$$\mathbf{z} = \left(\mathbf{W}^{(1)}\right)^T \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}, \tag{8.10}$$

resulting in the 3-by-1 vector $\mathbf{z}$. Then the nodes are activated using ReLU, that is $h_i = a(z_i) = \max\{0, z_i\}$.

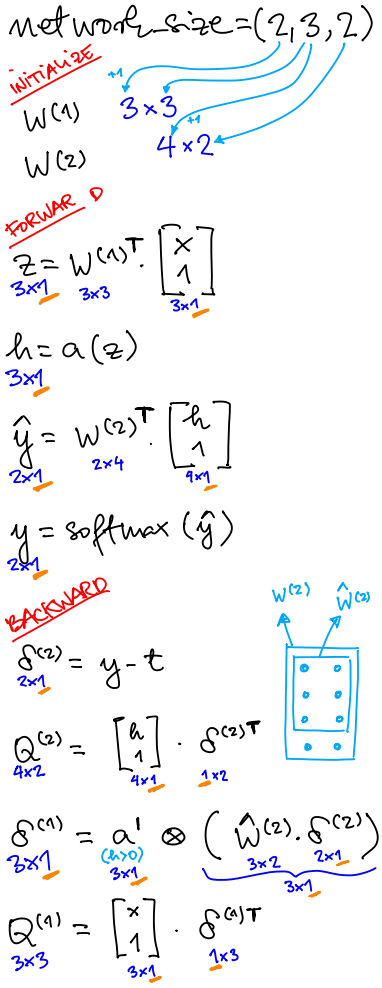

Figure 8.4: The sizes of the array when implementing the simple three-layer network as suggested in the exercise. Here, the input is just on 2D point. Later, when the network is to take mini-batches it will affect the underlined sizes.

From the hidden layer, the value $\hat{\mathbf{y}}$ is computed using a second vector-matrix multiplication

$$\hat{\mathbf{y}} = \left(\mathbf{W}^{(2)}\right)^T \begin{bmatrix} \mathbf{h} \\ 1 \end{bmatrix} , \tag{8.11}$$

resulting in a 2-by-1 vector. This is now activated using soft-max

$$y_j = \frac{\exp \hat{y}_j}{\sum_{k=1}^{2} \exp \hat{y}_k} . \tag{8.12}$$

You can implement the forward model as a function that takes the data $\mathbf{x}$ and weights $\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}$ as input. The function should return the predicted values $\mathbf{y}$, but also the values in the hidden nodes $\mathbf{h}$, because these will be used in the backpropagation. In the backpropagation, we will also need the derivative of the activation function for each neuron in the hidden layer. But those can be obtained from h_i since we know that we use ReLU activation, such that we have $a'(z_i) = 1$ if $h_i > 0$ and otherwise $a'(z_i) = 0$. In conclusion, you don't need to store the values of z_i.

Implementing backpropagation The backpropagation will run iteratively, where the weights $\mathbf{W}^{(2)}$ and $\mathbf{W}^{(1)}$ are updated in each iteration. You will start each iteration by running the forward pass and computing $\mathbf{y}$ and $\mathbf{h}$.

From $\mathbf{y}$ and target values, you can compute the cross entropy loss based on the current weights. It is a good idea to keep track of the loss when training the network. We expect the epoch loss (total loss for all points from the training set) to be decreasing.

We will now start from the output and compute the parameters needed to update the weights. First, we compute the 2-by-1 vector

$$\boldsymbol{\delta}^{(2)} = \mathbf{y} - \mathbf{t} . \tag{8.13}$$

where $\mathbf{t}$ is the target value. From $\boldsymbol{\delta}^{(2)}$ we can compute the partial derivatives needed for updating $\mathbf{W}^{(2)}$ as a matrix (outer) product

$$\mathbf{Q}^{(2)} = \begin{bmatrix} \mathbf{h} \\ 1 \end{bmatrix} \left(\boldsymbol{\delta}^{(2)}\right)^T . \tag{8.14}$$

Note here that $\mathbf{Q}^{(2)}$ is a 4-by-2 matrix – the same size as $\mathbf{W}^{(2)}$. You should not update the values of $\mathbf{W}^{(2)}$ before computing $\boldsymbol{\delta}^{(1)}$ in the next step.

We compute the values $\boldsymbol{\delta}^{(1)}$ for the the hidden layer as

$$\boldsymbol{\delta}^{(1)} = \mathbf{a}' \otimes \left(\hat{\mathbf{W}}^{(2)} \boldsymbol{\delta}^{(2)}\right) , \tag{8.15}$$

where $\hat{\mathbf{W}}^{(2)}$ is the 3-by-2 weight matrix based on $\mathbf{W}^{(2)}$ but with the last row (corresponding to bias) omitted. The vector $\mathbf{a}'$ is a 3-by-1 vector containing derivatives of the activation in the hidden layer, i.e. it has elements 0 or 1. The symbol $\otimes$ denotes element-wise multiplication.

We now move on to computing partial derivatives needed for updating $\mathbf{W}^{(1)}$ as a matrix (outer) product

$$\mathbf{Q}^{(1)} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \delta^{(1)} \, . \tag{8.16}$$

Now we update the weights

$$\mathbf{W}^{(1),\text{new}} = \mathbf{W}^{(1)} - \eta \mathbf{Q}^{(1)} \, , \tag{8.17}$$

$$\mathbf{W}^{(2),\text{new}} = \mathbf{W}^{(2)} - \eta \mathbf{Q}^{(2)} \, . \tag{8.18}$$

You can implement the backpropagation as a function that takes the data point $\mathbf{x}$, weight matrices $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$, target value $\mathbf{t}$, and learning rate η as input and returns the updated weight matrices.

Test your implementation The function `make_data.py` return points for training and test points on a regular grid. If you transform these points as you standardized the training points, you can pass the test points through the forward model, and use the output for illustrating the prediction by showing the result as an image.

Batch optimization A variant of stochastic gradient descent splits the training data into smaller subsets (minibatches) and updates weight according to the average of the gradients for the minibatch. One cycle through the entire training dataset is called a training epoch.

If you implement the forward model and backpropagation as described here using vector-matrix multiplications, only small modifications are required to be able to input a mini-batch of data points $\mathbf{x}$ and targets $\mathbf{t}$ as 2-by-m arrays. For this, you should remember to add values for bias nodes as 1-by-m of ones. Here, the forward model will return $\mathbf{y}$ as an m-by-2 array and $\mathbf{h}$ as an m-by-3 array. For the backward pass, the outer products when computing $\mathbf{Q}^{(l)}$ will give the sum of gradients, so you can choose to divide either the learning rate or the gradients with m.

Numeric stability Several steps in forward-backward passes may cause numerical problems, especially if the learning rate is high. If you experience such problems (`nan` or `inf` values populating your array), you should avoid division by zero in the computation of softmax or passing zero to logarithmic function in the computation of cross entropy. Also, exponential function in softmax may give infinity for large arguments.

One way of dealing with this is to ensure that predicted values y_i are all in some finite range, for example between 10^{-15} and 10^{15}.

8.4.3 *Variable number of layers and hidden units*

In the exercise above, you have obtained a neural network with a fixed architecture, i.e. the number of neurons and hidden layer. The architecture is however central in modeling with neural networks, and therefore you should make the number of layers and the number of neurons in each of the hidden layers a part of your input choice. You will also need this flexibility in the later exercises for classifying images, for example, the MNIST handwritten digits.

Suggested procedure

1. Implement a neural network keeping the single hidden layer and with a variable number of hidden units.
2. Extend this by implementing a neural network with a variable number of hidden layers and with a variable number of hidden units in each of these layers. The number of layers and hidden units are given as input to your function. Now you can play around with your model and from here it is easy to modify it to include other elements like other activation functions or minibatches.
3. Again test your implementation on the given data.

The expected output is shown in Figure 8.5. The coloring of the pixels in the background is obtained by passing points on the regular grid through the forward model and coloring the result in dark or bright gray depending on the classification result.

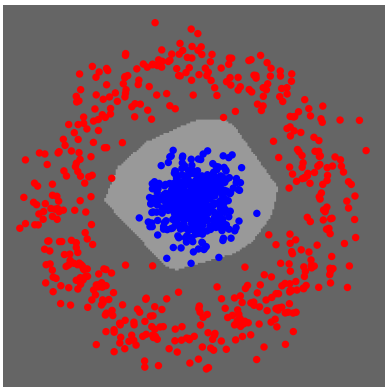

Figure 8.5: Result of training on the input data shown in colored points, and the test result is shown in the pixel colors in the background.