Towards high performance and efficient brain computer interface character speller : convolutional neural network based methods

Shan, H.

Chapter 2

Background

In this chapter, to better understand the contributions of this dissertation, we introduce some background information on machine learning, neural networks, Convolutional Neural Networks (CNNs), P300 signals, P300 spellers, the performance assessment of a P300 speller, and the datasets used in this dissertation.

In this dissertation, our proposed methods for P300-based BCIs are mainly based on CNNs. A CNN is a specific kind of a neural network. A neural network is a specific machine learning model in the machine learning field. Thus, first, we briefly introduce what machine learning is in Section 2.1. Then, we describe how a neural network works in Section 2.2. After that, we introduce what a CNN is in Section 2.3. The introductory text of each of the aforementioned sections is excerpts from well-known books with small modifications. For example, Section 2.1 is based on [Qiu], Section 2.2.1 and Section 2.2.2 are based on [Hay94], Section 2.2.3 and Section 2.3 are based on [Nie15].

The aim of this dissertation is to research and develop high performance and efficient P300-based BCI systems. Thus, we also introduce some background information on P300-based BCIs in Section 2.4.

Finally, in Section 2.5, we describe the datasets used in the dissertation to assess the performance of our proposed methods for P300-based BCIs.

2.1 Machine Learning

Machine learning is a subfield of artificial intelligence. Machine learning is the science of getting computers to learn from data in an autonomous manner [Sam67, M$^+$97].

Let us take an example to show how the machine learning works and also explain some other terminologies used in the machine learning field. Suppose that now we

need to select good apples in a fruit market. How does the machine learning works to select good apples?

First, we takes some apples from the market. We list 3 features: color, shape, and size of each apple. The color, shape, and size are called the features that are related to apples. Then, we mark each apple with labels. For example, the label for each apple can be the label "good" or the label "bad". Labeled features and their corresponding labels constitute a dataset. Typically, there are two kinds of dataset, i.e., training dataset and test dataset. A machine learning method learns from the training dataset. The test dataset is used to assess the performance of this machine learning method.

We use a 3-dimension vector $X = [x_1, x_2, x_3]$ to denote a vector constructed by the aforementioned apple's features, where x_1, x_2, and x_3 denote the color, shape, and size of an apple, respectively. Here X is called a feature vector. We use a 2-dimension vector $y = [y_1, y_2]$ to denote a vector constructed by the aforementioned labels for an apple, where y_1 denotes the label "good" and y_2 denotes the label "bad". We use D to denote a training dataset. D is shown in Equation (2.1), where F denotes that there are in total F apples, which means there are F feature vectors in the training dataset; $X^{(i)}, i \in \{1, ..., F\}$ denotes the ith feature vector in the training dataset, and $y^{(i)}, i \in \{1, ..., F\}$ denotes the corresponding label for $X^{(i)}$.

$$D = \{(X^{(1)}, y^{(1)}), (X^{(2)}, y^{(2)}), ..., (X^{(F)}, y^{(F)})\} \tag{2.1}$$

For the given training dataset D, we want to get a computer to automatically find a function $f(X, \theta)$ to build a mapping from the feature vector X to the label y, where θ is a set of parameters of the function $f(\cdot)$. The function $f(X, \theta)$ is called a machine learning model. By using an algorithm A, we can find a set of parameters θ^* that is able to make the function $f(X, \theta^*)$ build the mapping from the feature vector X to the label y from the training dataset D. This process is called learning or training. The algorithm A used in the learning or training process to find θ^* is called a learning algorithm.

After we find $f(X, \theta^*)$ from the training dataset, when we buy new apples next time, based on the features (that constitute the feature vector) of the new apples X^*, we can use the aforementioned trained model $f(X, \theta^*)$ to predict the labels (i.e., good apples or bad apples) for these new apples X^*.

To summarize the aforementioned introduction to the machine learning, we show the workflow of a machine learning method in Figure 2.1. This figure shows that the input to a machine learning method is a feature vector X, the output of this machine learning method is the label y. A machine learning model $f(X, \theta)$ is used in the machine learning method. By using a learning algorithm A and the training dataset D, the machine learning method finds a set of parameters θ^* that makes the function

$f(X, \theta^*)$ build the mapping from the feature vector X to the label y. After this, the machine learning method gets a trained model $f(X, \theta^*)$. Then, for a new input X^*, the trained model $f(X, \theta^*)$ can predict a label for this new input X^*.

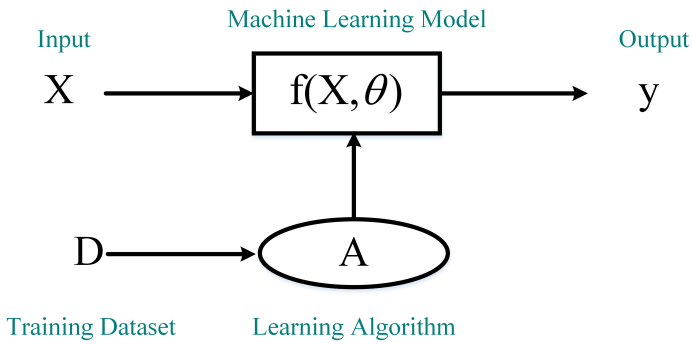

Figure 2.1: The workflow of machine learning.

2.2 Neural Network

A neural network is a specific machine learning model used in the machine learning method (introduced in Section 2.1). A neural network is made up of neurons . Therefore, we first introduce what a neuron is in a neural network in Section 2.2.1. Then, we describe how neurons constitute a neural network in Section 2.2.2. Finally, we introduce the learning algorithm used for neural networks.

2.2.1 Neurons

In this section, we introduce how a neuron works. First, we describe the model of a neuron in Section 2.2.1.1. Then, we introduce some functions used in the model of the neuron in Section 2.2.1.2.

2.2.1.1 Model of a Neuron

A neuron is an information processing unit which is fundamental to the operation of a neural network. Figure 2.2 shows the model of a neuron. We call this neuron neuron r since a neural network is constructed by many neurons (see Figure 2.4 and Figure 2.5 in Section 2.2.2). In Figure 2.2, neuron r takes several signals, namely, i_1, i_2, ..., i_m as inputs and produces a single output o_r. From this picture, we can see that a neuron has the following three basic elements:

1) A set of connecting links. Each of these links is characterized by a weight w_{rj}, $j \in [1, m]$. An input signal i_j is connected to neuron r by multiplying the weight w_{rj};

2) An adder. This adder sums the input signals (i_1, i_2, ..., i_m), weighted by the corresponding weights mentioned above;

3) An activation function. This activation function limits the amplitude of the output signal o_r to be in a certain range. Typically, the range of the output of a neuron is limited to $[0, 1]$ or $[-1, 1]$.

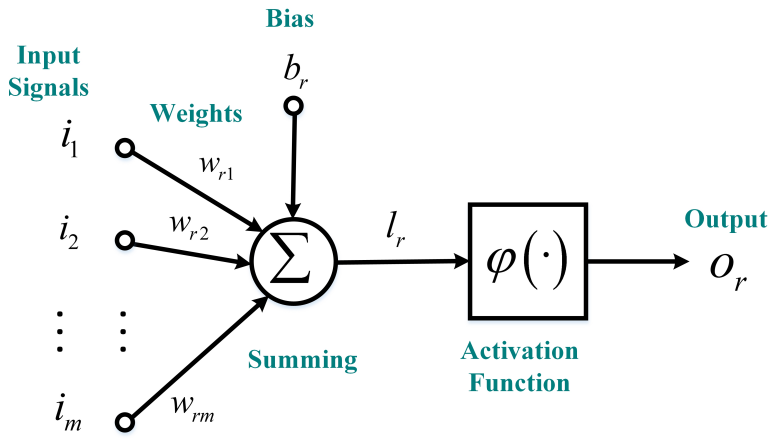

Figure 2.2: The model of a neuron.

Figure 2.2 shows that the model of a neuron also includes an externally applied parameter called bias. The bias is denoted by b_r in this model of neuron r. The bias b_r is used to increase or decrease the input of the activation function.

In mathematical terms, we can model neuron r by using Equations (2.2), (2.3) and (2.4), where i_1, i_2, ..., i_m are the input signals to neuron r; w_{r1}, w_{r2}, ..., w_{rm} are the weights of neuron r; b_r is the bias; $\varphi(\cdot)$ is the activation function; and o_r is the output of neuron r.

$$u_r = \sum_{j=1}^{m} w_{rj} i_j \tag{2.2}$$

$$l_r = u_r + b_r \tag{2.3}$$

$$o_r = \varphi(l_r) \tag{2.4}$$

2.2.1.2 Types of Activation Functions

The aforementioned activation function, denoted by $\varphi(\cdot)$, defines the output of neuron r. Here, we introduce some basic activation functions:

1) Threshold Function. The threshold function is given by Equation (2.5), where $\varphi(\cdot)$ denotes the activation function; and l_r denotes the input to the activation function $\varphi(\cdot)$, and l_r is defined using Equation (2.3).

$$\varphi(l_r) = \begin{cases} 1 & if \quad l_r \geq 0 \\ \\ 0 & if \quad l_r < 0 \end{cases} \tag{2.5}$$

2) Sigmoid Function. The sigmoid activation function is given by Equation (2.6), where a is used to control the output of the sigmoid function. Note that the output of a sigmoid function is in a continuous range $[0, 1]$ while the output of the threshold function is ether 1 or 0.

$$\varphi(l_r) = \frac{1}{1 + exp(-al_r)} \tag{2.6}$$

3) Rectified Linear Unit (ReLU) Function. The ReLU function is given by Equation (2.7). ReLU is by far the most commonly used activation function in CNNs.

$$\varphi(l_r) = max(0, l_r) \tag{2.7}$$

4) Softmax Function. The Softmax function is given by Equation (2.8), where p denotes that there are p neurons in total in a layer of a neural network. The layer of a neural network is described in Section 2.2.2.

$$\varphi(l_r) = \frac{e^{l_r}}{\sum_{n=1}^{p} e^{l_n}} \tag{2.8}$$

2.2.2 The Architecture of a Neural Network

Neurons constitute a neural network. In this section, we describe the architecture of a neural network that is constructed by the neurons. For better readability, we simplify the model of a neuron shown in Figure 2.2 with the graph shown in Figure 2.3. This means that the graph shown in Figure 2.3 denotes neuron r with input signals i_1, i_2,

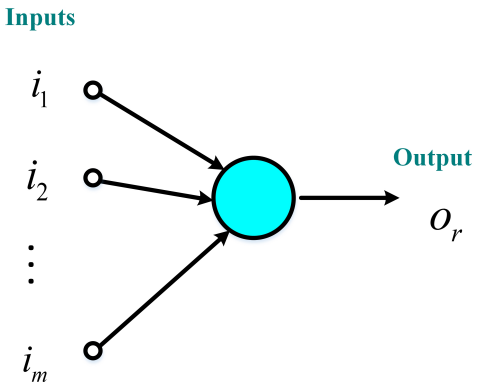

Figure 2.3: Architectural graph to model a neuron.

..., i_m and an output o_r. How neuron r works in this graph is given by Equations (2.2), (2.3) and (2.4) (For details please see Section 2.2.1.1).

In a neural network, the neurons are organized in the form of layers. In the simplest form of a neural network, we have an input layer of input signals that projects onto an output layer of neurons. We call this neural network the single-layer neural network. Here "single-layer" refers to the output layer of the network. We do not count the input layer of this network because there is no computation performed in the input layer. Figure 2.4 shows an example of a single-layer neural network. In this example, this single-layer neural network has four input signals and has one output layer of two neurons that produce outputs.

Typically, a neural network has more than one layer. Compared with a single-layer neural network, a neural network with more complex architecture has several layers of neurons. Such network is called a multi-layer neural network. In addition to the output layer and the input signals, a multi-layer neural network has one or more hidden layer of neurons. The function of the hidden layer of neurons is to intervene between the external input signals and the outputs of the network. By adding one or more hidden layers for the network, the network is enabled to extract higher-order features related to the input signals. The ability of extracting higher-order features by the hidden layers of neurons is particularly valuable when the size of the input layer is large. Figure 2.5 shows an example of a multi-layer neural network with one hidden layer and one output layer. In this example, this multi-layer neural network has 4 input signals, one hidden layer with 3 neurons, and the output layer with 2 neurons. The input signals are the input to the hidden layer of this network. The output signals of this hidden layer are the input to the output layer of the network. Typically, the input to the neurons in each layer of a multi-layer neural network is the output signals of its

preceding layer only. The neural network shown in Figure 2.5 is also called a fully-connected neural network in the sense that every node in each layer of the network is connected to every other node in the adjacent forward layer. Here a node denotes an input signal or a neuron in the graph shown in this figure.

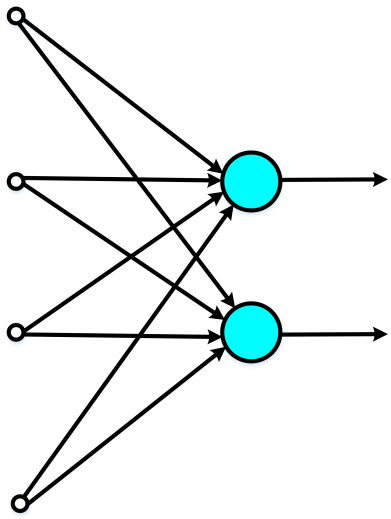

Figure 2.4: An example of a single-layer neural network.

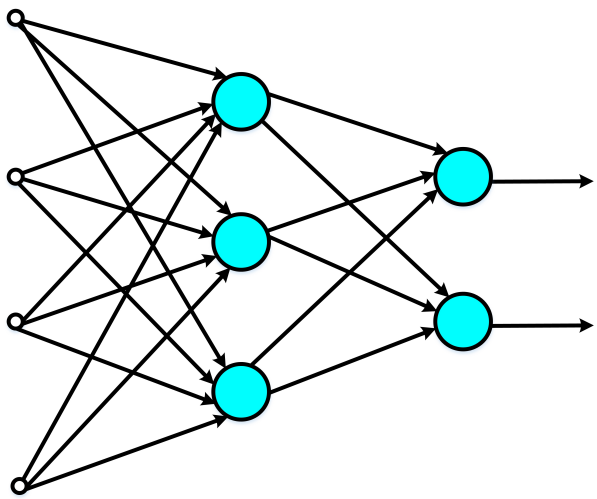

Figure 2.5: An example of a multi-layer neural network with one hidden layer and one output layer.

2.2.3 Learning Process of a Neural Network

As discussed in Section 2.1, when we use a neural network as a machine learning model, we need to use a learning algorithm to find a set of parameters, i.e., the weights and biases of all neurons in the neural network, that make this neural network be able to map input X to label y in the training dataset. However, the learning algorithm is not able to calculate the perfect weights and biases for a neural network. Instead, the learning process of a neural network is regarded as an optimization problem, where the learning algorithm is used to explore the space of possible sets of weights and biases for the neural network. We use a function to evaluate a candidate solution (i.e. a set of weights and biases for the neural network). This function is called a cost function or a loss function. For example, we can define a cost function as given in Equation (2.9). In this equation, X denotes the input vector in the training dataset and $X^{(i)}$ is the ith input feature vector in the training dataset. y denotes the desired output of the network and $y^{(i)}$ is the desired output of the network when the input to the network is $X^{(i)}$. $f_{NN}(\cdot)$ denotes the neural network. W denotes all the weights in the neural network. B denotes all the biases in the network. F denotes the total number of input feature vectors. This loss function is called the Mean Squared Error (MSE) cost function. From this cost function, we can see that $C(W, B)$ is non-negative. When the cost $C(W, B)$ becomes very small, i.e., $C(W, B)$ is close to 0, $f_{NN}(X, W, B)$ is approximately equal to y. This means that the learning algorithm has found very good weights W and biases B such that the neural network with these weights and biases approximately maps the input of this network X to the desired output of the network y. In contrast, when the cost $C(W, B)$ is large, this means that $f_{NN}(X, W, B)$ is not equal to y, showing that our neural network with the weights W and the biases B cannot map well the input of this network X to the output of the network y. Now, the cost function that is commonly-used for a neural network is called the cross-entropy cost function. The cross-entropy cost function is given in Equation (2.10). In this equation, E denotes that there are E neurons in the output layer of a neural network. $y_j^{(i)}$ denotes the desired output of the jth neuron in the output layer of a neural network when the input to the network is $X^{(i)}$. $f_{NN_j}(X^{(i)}, W, B)$ denotes the actual output of the jth neuron in the output layer of a neural network when the input to the network is $X^{(i)}$.

$$C(W, B) = \frac{1}{F} \sum_{i=1}^{F} \|y^{(i)} - f_{NN}(X^{(i)}, W, B)\|^2 \tag{2.9}$$

$$C(W, B) = -\frac{1}{F} \sum_{i=1}^{F} \sum_{j=1}^{E} [y_j^{(i)} log(f_{NN_j}(X^{(i)}, W, B)) + (1 - y_j^{(i)}) log(1 - f_{NN_j}(X^{(i)}, W, B))] \tag{2.10}$$

From the aforementioned description, we can see that the objective of the learning algorithm is to minimize the cost $C(W, B)$. More specifically, we seek to find a set of weights W and biases B that minimize this cost as much as possible. For better readability, we use $C(v)$ to denote a cost function. $C(v)$ can be a function of many parameters such as $v = v_1, v_2, ..., v_h$. Suppose C is a function of just two parameters v_1, v_2. Figure 2.6 shows an example of function C with v_1 and v_2. Our object is to find v_1, v_2 where C achieves its minimum. For the simple function shown in Figure 2.6, we can use calculus to try to find the minimum analytically. We could compute derivatives and then try using them to find places where C is an extremum. However, the cost function of a neural network can have many more parameters and be much more complex because a neural network contains much more parameters, i.e., the weights and biases of all neurons in the network. For example, very large neural networks have cost functions that depend on billions of weights and biases. It is impossible to use calculus to minimize the cost function.

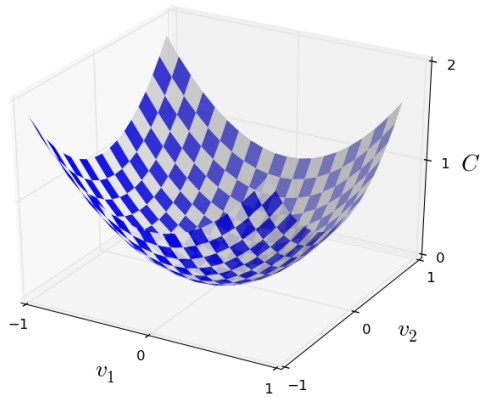

Figure 2.6: An example of a cost function C with two parameters v_1 and v_2.

To solve the aforementioned minimization problem, we can use a method, called gradient descent. We use an analogy to explain how the gradient descent method works to solve this minimization problem. This analogy is shown in Figure 2.7. As shown in this figure, we can think of our cost function as a kind of a valley and imagine a ball rolling down the slope of the valley. When the ball reaches the bottom of this valley, this means we find the minimum of the cost function. Then, the problem comes to how we make a rule that makes the ball roll down to the bottom of the valley. We can use the calculus to describe the move of a ball with a small amount $\triangle v_1$ in the v_1

direction and a small amount $\triangle v_2$ in the v_2 direction by using Equation (2.11). We need to find $\triangle v_1$ and $\triangle v_2$ that make $\triangle C$ negative (negative $\triangle C$ means that the ball is rolling down into the valley). We define that $\triangle v$ is equal to $(\triangle v_1, \triangle v_2)^T$ as shown in Equation (2.12), where T is the transpose operation that turns row vectors into column vectors in a matrix. We define that the gradient of C, denoted by $\bigtriangledown C$, is equal to the vector of partial derivatives $(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$, as shown in Equation (2.13). With these definitions, Equation (2.11) can be rewritten to be Equation (2.14). In order to make $\triangle C$ negative, we can make $\triangle v$ to be equal to $-\eta \bigtriangledown C$ as shown in Equation (2.15), where η is a small, positive parameter, called the learning rate. Then, by combing Equation (2.14) and Equation (2.15), $\triangle C = -\eta \bigtriangledown C \cdot \bigtriangledown C = -\eta \| \bigtriangledown C\|^2$. Because $\| \bigtriangledown C\|^2 \geq 0$ and $\eta > 0$, $\triangle C \leq 0$. This means that C will always decrease and never increase. Thus, we use Equation (2.15) to define the rule of how to move the ball in the gradient descent algorithm. This means that we use Equation (2.15) to compute a value $\triangle v$ and then move the position of the ball v to a new position v' by the amount of $\triangle v$, as shown in Equation (2.16). Then we will use updated rule again to make another movement of the ball. By keeping doing this, we can decrease C until we reach the (approximate) minimum of the cost function C.

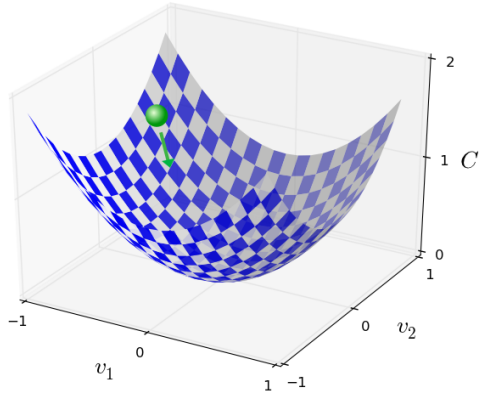

Figure 2.7: The analogy of using gradient descent to minimize a cost function.

$$\triangle C \approx \frac{\partial C}{\partial v_1} \triangle v_1 + \frac{\partial C}{\partial v_2} \triangle v_2 \tag{2.11}$$

$$\triangle v = (\triangle v_1, \triangle v_2)^T \tag{2.12}$$

$$\bigtriangledown C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T \tag{2.13}$$

$$\triangle C \approx \bigtriangledown C \cdot \triangle v \tag{2.14}$$

$$\triangle v = -\eta \bigtriangledown C \tag{2.15}$$

$$v \rightarrow v' = v - \eta \bigtriangledown C \tag{2.16}$$

The aforementioned discussion describes how the gradient descent method works when the cost function C has two parameters. When C is a function of h parameters, i.e., v_1, v_2, ..., v_h, $\bigtriangledown C$ is calculated using Equation (2.17). We repeatedly apply the rule shown in Equation (2.18) until we reach the (approximate) minimum of the cost function C.

$$\bigtriangledown C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, ..., \frac{\partial C}{\partial v_h})^T \tag{2.17}$$

$$v \rightarrow v' = v - \eta \bigtriangledown C \tag{2.18}$$

In fact, for a neural network, v is constituted by all weights W=w_1, w_2, ..., w_d and all biases B=b_1, b_2, ..., b_g. Therefore, to update the weights of a neural network, we repeatedly apply the rule shown in Equation (2.20) to reach the (approximate) minimum of the cost function C. In Equation (2.20), $\bigtriangledown C_W$ is calculated using Equation (2.19). Also, to update the biases of a neural network, we repeatedly apply the rule shown in Equation (2.22) to reach the (approximate) minimum of the cost function C. In Equation (2.22), $\bigtriangledown C_B$ is calculated using Equation (2.21).

$$\bigtriangledown C_W = (\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, ..., \frac{\partial C}{\partial w_d})^T \tag{2.19}$$

$$W \rightarrow W' = W - \eta \bigtriangledown C_W \tag{2.20}$$

$$\bigtriangledown C_B = (\frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial b_2}, ..., \frac{\partial C}{\partial b_g})^T \tag{2.21}$$

$$B \rightarrow B' = B - \eta \bigtriangledown C_B \tag{2.22}$$

Now, researchers often use the gradient descent method with momentum, which is called the momentum-based gradient descent method. The momentum technique modifies the gradient descent method in two ways. Firstly, the momentum technique introduces a notion of "velocity" for the parameters we optimize. The gradient descent method changes the "velocity" of the parameters, not (directly) the "position" of the parameters, and only indirectly affects the "position" of the parameters. Secondly, the momentum technique introduces a friction term, which can gradually reduce the "velocity" of the parameters. In mathematical terms, the momentum-based gradient descent method replaces the updating rule for W (given in Equation (2.20) used in the gradient descent method without momentum) with a new updating rule given in Equation (2.23), and (2.24), where V_w denotes the aforementioned "velocity" for W, and μ denotes the aforementioned friction term and is called the momentum parameter. Also, the momentum-based gradient descent method replaces the updating rule for B (given in Equation (2.22) used in the gradient descent method without momentum) with a new updating rule given in Equation (2.25), and (2.26), where V_b denotes the aforementioned "velocity" for B.

$$V_w \rightarrow V_w' = \mu V_w - \eta \bigtriangledown C_W \tag{2.23}$$

$$W \rightarrow W' = W + V_w' \tag{2.24}$$

$$V_b \rightarrow V_b' = \mu V_b - \eta \bigtriangledown C_B \tag{2.25}$$

$$B \rightarrow B' = B + V_b' \tag{2.26}$$

One problem of the learning process of a neural network is called overfitting. Overfitting happens when a neural network learns the details of the training data to the extent that it negatively impacts the performance of this neural network on new data. This means that random fluctuations in the training data is learned by the neural network. Unfortunately, the learned random fluctuations cannot apply on new data, thereby negatively impacting the generalizing ability of the network.

In order to reduce the overfitting, one commonly-used technique, called weight decay, is utilized. The weigh decay technique modifies the updating rule for weights W and does not change the updating rule for biases B in the gradient descent method. The gradient descent method with weight decay replace the updating rule for W (given in Equation (2.20) used in the gradient descent method without weight decay) with a new updating rule given in Equation (2.27). In Equation (2.27), λ is called the weight decay parameter; F denotes the total number of input feature vectors and η is the learning rate. The updating rule for B used in the gradient descent method with weight decay is the same as the updating rule for B (given in Equation (2.22)) used in the gradient descent method without weight decay.

$$W \rightarrow W' = (1 - \frac{\eta\lambda}{F})W - \eta \bigtriangledown C_W \tag{2.27}$$

2.3 Convolutional Neural Network

Convolutional Neural Network (CNN) is a specific kind of neural network. In recent years, CNNs have been the most commonly-used neural networks to recognize images.

When using a fully-connected neural network (introduced in Section 2.2.2) to recognize images, the fully-connected neural network has the problem that it uses a large number of parameters to recognize images. Before introducing this problem, let us first describe how to use a neural network to recognize images. We take the handwritten digit recognition as an example of image recognition. The handwritten digit recognition is to recognize what digit (e.g., 1,3, 6,...) is for an image of a handwritten digit. From the discussion in Section 2.1 and Section 2.2.3, we can see that when using the machine learning method to recognize a handwritten digit, we need to develop a machine learning model. Here, we use a neural network, denoted by $f_{NN}(X, W, B)$, as a machine learning model, where f_{NN} denotes a neural network; W denotes all the weights in the neural network; B denotes all the biases in the network. X is called

a tensor and denotes the inputs to the neural network. For example, if X is an image with 640 $\times$ 480 pixels, we call X a (640 $\times$ 480) tensor. We train this neural network $f_{NN}(\cdot)$ with the training dataset that consists of handwritten digits with their corresponding labels (e.g., 1, 3, 6). As introduced in Section 2.2.3, the gradient descent is used to find the weights W and biases B that make the neural network $f_{NN}(X, W, B)$ build the (approximate) mapping from the handwritten digits to the labels. Then, for a new handwritten digit, the trained neural network can predict a label for this new handwritten digit.

Up to this point, we have known how to use a neural network to recognize images of handwritten digits. Now let us introduce the reason of why the fully-connected neural network has the problem of using a large number of parameters to recognize images. In the aforementioned example of the handwritten digit recognition, the input is an image of a handwritten digit. This image is 28 $\times$ 28 pixel image. This means that the number of the input signals in the input layer of a fully-connected neural network is 784 = 28 $\times$ 28. Suppose a simple fully-connected neural network which architecture is a 2-layer network with one hidden layers and one output layer. We suppose that each hidden layer has 10 neurons and the output layer of this neural network has 10 neurons. The number of parameters (i.e., all weights and biases) of this neural network is (784 $\times$ 10+10) + (10 $\times$ 10 + 10) = 7960. Unfortunately, such simple fully-connected neural network cannot work well to recognize handwritten digits. Suppose a more complex fully-connected neural network which architecture is a 3-layer network with two hidden layers and one output layer. Each hidden layer of this network has 50 neurons, and the output layer still has 10 neurons. The number of parameters of this network is 42290. From this example, we can see that with the increase of the number of hidden neurons and hidden layers, the number of the parameters of a fully-connected layer is dramatically increased. This means that when we design a fully-connected neural network that can be useful for image recognition, the number of the parameters of such network will be large. The large number of parameters dramatically increases the time of training such fully-connected neural network because in the training process, the gradient descent algorithm will need quite a long time to find a large number of parameters that make this network (approximately) maps the handwritten digits to the labels (For details of the training process of a neural network, please see Section 2.2.3).

To address the aforementioned problem of the fully-connected neural network, Convolutional Neural Network is proposed to recognize images. The name "Convolutional Neural Network" indicates that this neural network utilizes a mathematical operation called convolution. We will introduce what the convolution operation is in Section 2.3.1. Then, we introduce the characteristics of a CNN in Section 2.3.2. Finally, we introduce the architecture of a CNN in Section 2.3.3.

2.3.1 The Convolution Operation

The convolution operation is an important operation in analytical mathematics. In this section, we introduce the 2-dimension convolution operation because the 2-dimension convolution operation is widely used for image recognition and also used in our proposed CNN-based methods for P300-based BCIs in this dissertation.

The 2-dimension convolution operation is defined by Equation (2.28), where $\otimes$ denotes the convolution operation. Z, X, and K are 2-dimension matrices. X denotes the input matrix to the convolution operation; Z denotes the outputs of the convolution operation; K is called the kernel of the convolution operation. (k_1, k_2) is called the kernel size. (s_1, s_2) is called the stride. In Equation (2.28), $Z(i, j)$ denotes the datum in the ith row and the jth column of the matrix Z; $X(i, j)$ denotes the datum in the ith row and the jth column of the matrix X; and $K(m, n)$ denotes the datum in the mth row and the nth column of the matrix K.

$$\begin{aligned} Z(i,j) &= (X \otimes K)(i,j) \\ &= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} X((i-1)s_1 + 1 + m, (j-1)s_2 + 1 + n) K(m,n) \end{aligned} \tag{2.28}$$

2.3.2 The Characteristics of Convolutional Neural Network

A CNN has three characteristics, i.e., the receptive field, the parameter sharing, and the pooling layers. We introduce these three characteristics in turn with the following.

1) **Receptive Field**. Firstly, we introduce what a receptive field in a CNN is. We still take the handwritten digit recognition as an example to describe the receptive field. When using a CNN to recognize the handwritten digits, the input to a CNN is a 28 $\times$ 28 pixel image. Let us take this input as a 28 $\times$ 28 square of input neurons. The value of each input neuron is the value of the corresponding pixel in the image. We take a small region of the input neurons to connect each hidden neuron in the first hidden layer of the CNN. For example, we can take a 5 $\times$ 5 region, containing 25 input neurons that corresponding to 25 input pixels, to connect to each hidden neuron in the hidden layer of the CNN. The region, which contains several neurons in a layer and is connected to a neuron in the next layer, is called the receptive field. Each connection which connects the receptive field with the hidden neuron represents a weight. One hidden neuron uses one bias. In this way, a hidden neuron is used to learn the features from a receptive field of input neurons. We then slide the receptive field across the entire input image. For example, first, the receptive field in the top-left corner of the input image is connected to the top-left hidden neuron in the first hidden layer. Then, we slide the receptive field over by one pixel (i.e., by one input neuron) to the right

to connect to the second hidden neuron in the first hidden layer. For each receptive field, there is a corresponding hidden neuron connected with this receptive field in the hidden layer. In this way, we build a hidden layer, and this hidden layer is called a feature map.

2) **Parameter Sharing**. Secondly, we introduce the parameter sharing in the CNN. Here, the parameter denotes the weights and biases used in a CNN. As described above, each hidden neuron in the hidden layer uses one biases and some wights that are connected to the corresponding receptive field. The parameter sharing means that all the neurons in a hidden layer of a CNN use the same weights and bias. The characteristic of parameter sharing in CNN is able to significantly reduce the number of parameters. Thus, the parameter sharing can solve the problem in the fully-connected neural networks, i.e., the fully-connected neural networks use a large number of parameters when used to recognize a image.

3) **Pooling Layers**. Finally, we introduce the pooling layers in the CNN. Pooling layers are often used after the convolution layers. The pooling operation in a pooling layer converts each feature map from the convolution layer into a condensed feature map by summarizing a region of neurons in the feature map. For example, the pooling operation, called max-pooling, takes the maximum in a region of neurons as a unit in the pooling layer. A CNN often uses more than one feature map, and the pooling operation is applied to each feature map, separately. Thus, if there are 5 feature maps, the pooling layer will output 5 condensed feature maps.

2.3.3 The Architecture of Convolutional Neural Network

After introducing the characteristics of the CNN, we describe the architecture of a CNN. The architecture of a CNN consists of three kinds of layers, i.e., convolution layers, pooling layers and fully-connected layers. A convolution layer performs the convolution operation introduced in Section 2.3.1. A pooling layer performs the pooling operation described in Section 2.3.2. A fully-connected layer performs the fully-connected operation shown in Section 2.2.2. Figure 2.8 shows an example of the architecture of a CNN used to recognize the handwritten digits. In this example, the input to the CNN is a 28×28 pixel image of a handwritten digit. The CNN has one convolution layer, one pooling layer, and one fully-connected layer. The input to the convolution layer is 28×28 input neurons. The convolution layer outputs 3 feature maps. These three feature maps are the input to the pooling layer. The pooling layer apply the pooling operation and outputs 3 condensed feature maps. These condensed feature maps are the input to the fully-connected layer. This fully-connected layer has 10 neurons that correspond to the 10 possible labels (e.g., 0, 1, 2, 3, ..., 9) of the handwritten digits. This fully-connected layer connects each of the 10 neuron with the condensed feature maps generated from the pooling layer.

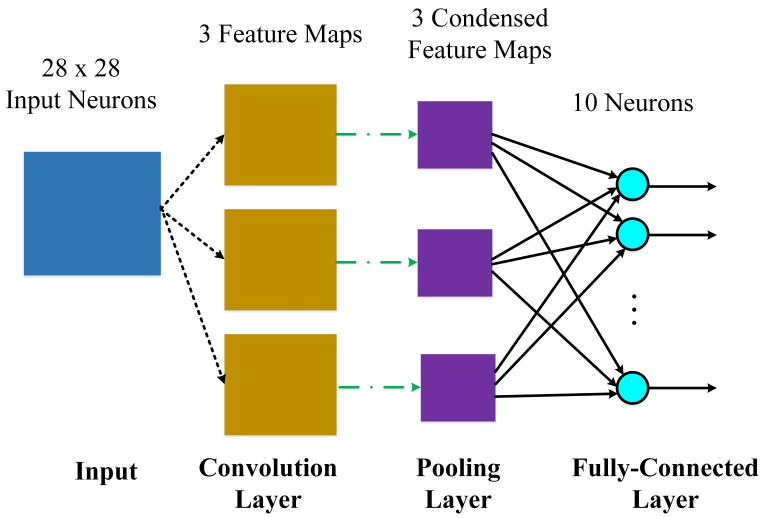

Figure 2.8: An example of the architecture of a CNN used for the handwritten digit recognition.

2.4 P300-based Brain Computer Interface

In this dissertation, we focus on P300-based BCIs. The P300 signal is the target signal used in a P300-based BCI and the P300 speller is the benchmark and the most commonly-used application of a P300-based BCI. First, we introduce some background information on the P300 signal and the P300 speller in Section 2.4.1 and Section 2.4.2, respectively. Then, we describe the metrics to assess the performance of the P300 speller in Section 2.4.3.

2.4.1 P300 Signal

Chapman and Bragdon first discovered the P300 signal in 1964 [CB64]. The P300 signal is a kind of an event-related potential (ERP). The P300 signal, recorded in EEG, occurs with a positive deflection in voltage at a latency about 300ms after a rare stimulus, as shown in Figure 2.9. The P300 signal is also called P3 wave because it is the third major positive peak in the late sensory and the late positive component [Pic92].

The P300 signal is an endogenous evoked potential because the evoking of a P300 signal does not have any relationship with the physical attributes of a stimulus, but has a relationship with a subject's (person's) reaction to the stimulus. The P300 signal is usually elicited using the oddball paradigm. In this paradigm, the target stimulus appears in low probability while the non-target stimulus appears in high probability.

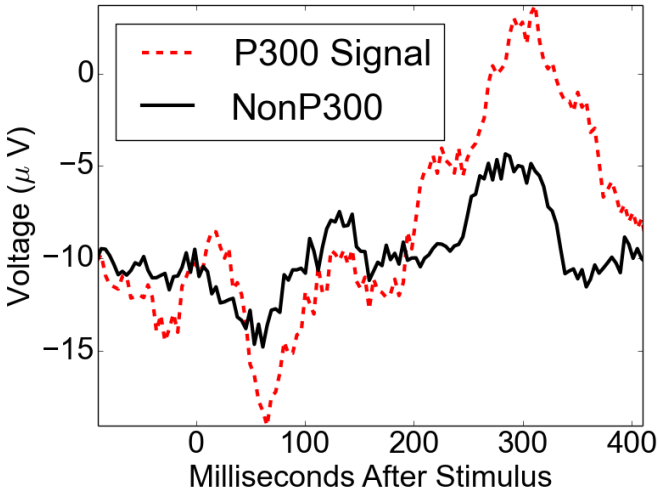

Figure 2.9: P300 signal.

The subject in this paradigm is detecting a rare target stimulus among the non-target stimuli. The P300 signal is only able to be evoked in the subject's brain when this subject detects the rare target stimulus.

The P300 signal is typically measured most strongly by the electrodes covering the parietal lobe. The amplitude of a P300 signal varies with the rareness of the target stimulus. The latency of a P300 signal varies with the difficulty of discriminating the target stimulus from the non-target stimuli. For example, the typical latency of a P300 signal evoked in a young healthy adult is about 300ms while the latency of a P300 signal evoked in subjects (persons) with decreased cognitive ability is longer than 300ms. Due to its reproducibility and ubiquity, the P300 signal is a common choice for psychological tests in both the clinic and laboratory.

From the aforementioned description of the P300 signal, we can infer when the subject detects a target stimulus by detecting the evoked P300 signal in the subject's brain signals. The detection of P300 signals from brain signals can be considered as a binary classification problem. There are two classes in this classification problem: one class corresponds to the presence of a P300 signal within a certain time period while the second class corresponds to the absence of a P300 signal within the time period. If we use E to denote a classifier (e.g., a CNN as introduced in Section 2.3) to classify the P300 signal, and X to denote the subject's brain signals within a certain time period, the P300 signal detection process can be expressed as Equation (2.29). In this equation, P^1 denotes the probability, predicted by this classifier, of having a P300 signal in this time period, P^0 denotes the probability, predicted by this classifier,

of not having a P300 signal in this time period.

$$E(X) = \begin{cases} 1 & if \quad P^1 > P^0 \\ 0 & otherwise \end{cases} \tag{2.29}$$

2.4.2 P300 Speller

Farwell and Donchin developed the first P300-based BCI character speller in 1988 [FD88]. The subject in the experiment is presented with a 6 by 6 character matrix (see Figure 2.10) and he focuses his attention on a target character he wants to spell. All rows and columns in this matrix are intensified successively and randomly but separately. Each row or column intensification lasts for time period t_1, followed by a blank time period of the matrix t_2. Two out of twelve intensifications contain the target character, i.e., one target row and one target column. As a result, the target row/column intensification becomes a rare stimulus to the subject. A P300 signal is then evoked by this rare stimulus. By detecting the P300 signal, we can infer which row or column the subject is focused on. By combing the row and column positions, we can infer the target character position. After the inference of one character, the matrix is blank for time period t_3 to inform the subject that the current character is completed and to focus on the next character.

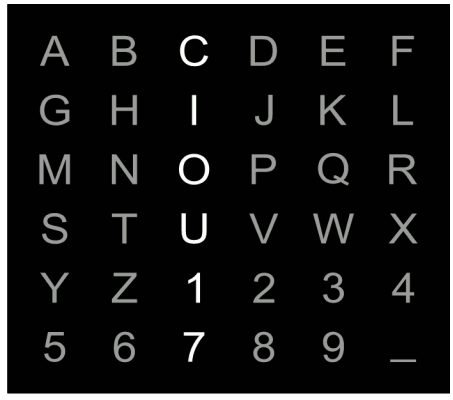

Figure 2.10: P300 speller character matrix.

Assume that one epoch includes 12 intensifications, in which there exist one target row intensification and one target column intensification. Then, in theory, one epoch is sufficient to infer one target character. However, in practice, since the P300 signal has a very low Signal to Noise Ratio (SNR) and is also influenced by artifacts, one epoch can hardly be sufficient to infer one target character correctly. As a

result, in practice, experimenters use many epochs to help the subject spell one character. The detailed calculation for determining the position of the target character is given in Equations (2.30), (2.31), and (2.32), where $P^1_{(i,j)}$ denotes the probability of the presence of a P300 signal in the jth intensification and the ith epoch, $Sum_{(j)}$ denotes the sum of the probabilities for the jth intensification when using k epochs, $index_{col}$ denotes the column index of the target character in the matrix in Figure 2.10, and $index_{row}$ denotes the row index of the target character. When $j \in [1, 6]$, j denotes a column intensification. When $j \in [7, 12]$, j denotes a row intensification. Equation (2.30) cumulates the probabilities of having a P300 signal evoked by intensification j over k epochs. In Equation (2.31), we assign the index of the maximum $Sum_{(j)}$ to $index_{col}$ when $j \in [1, 6]$. This equation finds the index of the column intensification, with the maximum sum of probabilities, to have evoked a P300 signal. This index is the column position of the target character when using k epochs. In Equation (2.32), the row position of the target character when using k epochs is calculated in the same way as in Equation (2.31). The position of the target character in the matrix in Figure 2.10 is the coordinate formed by the target row position and the target column position.

$$Sum_{(j)} = \sum_{i=1}^{k} P^1_{(i,j)} \tag{2.30}$$

$$index_{col} = \underset{1 \le j \le 6}{argmax} \{Sum_{(j)}\} \tag{2.31}$$

$$index_{row} = \underset{7 \le j \le 12}{argmax} \{Sum_{(j)}\} \tag{2.32}$$

2.4.3 Performance Assessment of P300 Speller

As indicated in Chapter 1, in this dissertation, we use the P300 speller as the benchmark application of a P300-based BCI. As the benchmark application of a P300-based BCI, we need metrics to assess the performance of the P300 speller. More specifically, we need metrics to assess the communication accuracy and the communication speed of the P300 speller. As typically done in related research works for the P300 speller, we use the character spelling accuracy to assess the communication accuracy of the P300 speller as well as we use the Information Transfer Rate (ITR) to assess the communication speed of the P300 speller.

To calculate the character spelling accuracy of the P300 speller, we use Equation (2.33). In this equation, $acc_{char(k)}$ denotes the character spelling accuracy when using k epochs for each character (see Section 2.4.2), $N_{tc(k)}$ denotes the number of

correctly inferred characters when using k epochs for each character, and S_c denotes the number of all characters.

$$acc_{char(k)} = \frac{N_{tc(k)}}{S_c} \tag{2.33}$$

In addition to using the character spelling accuracy to assess the communication accuracy of the P300 speller, we also use the Information Transfer Rate (ITR) for the assessment of the communication speed of the P300 speller. ITR has been the most commonly applied metric to assess the communication speed of P300-based BCIs [WW12, LWG16, NRS17, IKV18]. ITR has been introduced by Shannon and Weaver [SW49]. It is calculated by Equation (2.34) [WRMP98], where $acc_{char(k)}$ is calculated using Equation (2.33) and N_{cla} is the number of classes. Here, we have 36 characters to spell (see Figure 2.10), so N_{cla} =36. T_k denotes the time needed to spell a character when using k epochs. T_k is calculated using Equation (2.35), where t_1, t_2, and t_3 are the time periods described in the first paragraph of Section 2.4.2. For more detailed explanation of Equation (2.34), please refer to [WRMP98].

$$ITR_k = \frac{60(acc_{char(k)} \log_2(acc_{char(k)}) + (1 - acc_{char(k)}) \log_2(\frac{1-acc_{char(k)}}{N_{cla}-1}) + \log_2(N_{cla}))}{T_k} \tag{2.34}$$

$$T_k = t_3 + 12 \times (t_1 + t_2) \times k \quad 1 \leq k \leq 15 \tag{2.35}$$

Here, we calculate the theoretical maximum ITR when the datasets described in Section 2.5 are used in this dissertation because we want to compare the ITR achieved by our methods with the theoretical maximum ITR. When using the datasets described in Section 2.5, N_{cla}=36, t_1=100ms, t_2=75ms, and t_3=2.5s (for details please refer to Section 2.5). The theoretical maximum ITR is achieved when we use the least time to spell a character and achieve the highest spelling accuracy. The least time to spell a character means that we use only one epoch to spell a character. i.e., k=1. Thus, the least time we use to spell a character is T_1= 2.5s + 12 $\times$ (0.1s+0.075s) = 4.6s. The highest spelling accuracy is 100% (i.e., $acc_{char(1)}$ = 1). As a result, the theoretical maximum ITR when using the datasets described in Section 2.5 is $\frac{60 \log_2(36)}{4.6}$ bits/min = 67.43 bits/min.

2.5 Datasets

This dissertation uses three benchmark datasets, namely, BCI Competition II - Data set IIb [Bla03] as well as BCI Competition III - Data set II Subject A and Subject

B [Bla08]. Since many methods for P300-based BCIs use these three benchmark datasets, we can fairly compare the character spelling accuracy and ITR, introduced in Section 2.4.3, achieved by our CNN-based P300 speller with the character spelling accuracy and ITR achieved by other state-of-the-art methods for the P300 speller. Here, we give a short description of the three datasets.

BCI Competition II - Data set IIb and BCI Competition III - Data set II Subject A and Subject B are provided by the Wadsworth Center, NYS Department of Health. They are recorded with the BCI2000 platform [SMH+04], using the P300 speller described in Section 2.4.2. EEG signals are collected from 64 sensors at a sampling frequency of 240Hz. One intensification lasts for time period t_1=100ms, followed by a blank time period of the matrix t_2=75ms. The experiment uses 15 epochs for each character. Each character epoch is represented by 12 sets of signal samples. One set of signal samples, as shown in Figure 2.11, is a (N, C) matrix. In this matrix, each row has $N = T_s \times F_s$ (F_s is the signal sampling frequency) signal samples in the time period between 0 and T_s posterior to the beginning of each intensification, and each column has the signals samples taken at the same time from all C sensors used in the EEG headset. After each sequence of 15 epochs, the matrix is blank for time period t_3=2.5s to inform the subject that the current character is completed and to focus on the next character.

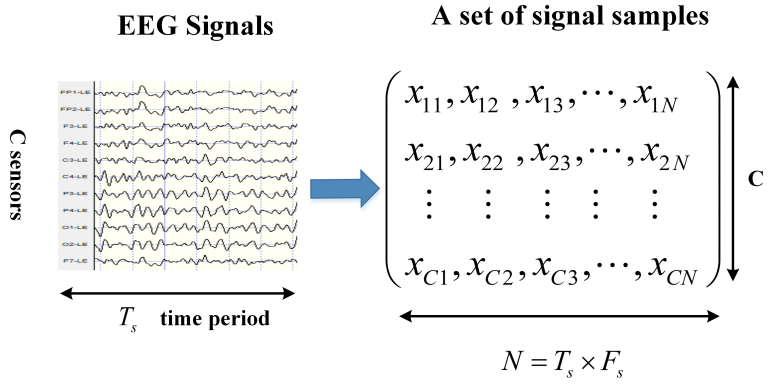

Figure 2.11: An example of a set of signal samples, where F_s is the signal sampling frequency

In BCI Competition II - Data set IIb, there is one subject with separate training and test datasets. The training dataset has 42 characters and the test dataset has 31 characters. In each character epoch, represented by 12 sets of signal samples, 2 sets have a P300 signal and 10 sets do not have a P300 signal. So, the training dataset has 42 $\times$ 15 $\times$ 2 = 1260 sets of signal samples labelled “P300”, and there are 42 $\times$ 15 $\times$

10 = 6300 sets labelled “non-P300”. The test dataset has 930 sets of signal samples labelled “P300” and 4650 sets labelled “non-P300”.

In BCI Competition III - Data set II, there are two subjects. We call them Subject A and Subject B. For each subject, the training dataset has 85 characters and the test dataset has 100 characters. So, the training dataset has 2550 sets of signal samples labelled “P300” and 12750 sets labelled “non-P300”. The test dataset has 3000 sets of signal samples labelled “P300” and 15000 sets labelled “non-P300”.

Table 2.1 shows the number of P300s/non-P300s for each dataset. II denotes BCI Competition II - Data set IIb, III-A denotes BCI Competition III - Data set II Subject A, and III-B denotes BCI Competition III - Data set II Subject B.

Table 2.1: Number of P300s/non-P300s for each dataset.

Dataset	Train		Test	
	P300	non-P300	P300	non-P300
II	1260	6300	930	4650
III-A	2550	12750	3000	15000
III-B	2550	12750	3000	15000