Artificial Neural Networks

Shintaro Hagiwara, Patrick Boily [shintaro.hagiwara@cqads.carleton.ca]
Centre for Quantitative Analysis and Decision Support, Carleton University, Ottawa, Ontario, Canada

1 Introduction

What do you feel like eating for dinner tonight? Do you feel like pizza, pasta, or perhaps something Asian like sushi? When you are asked such a question, you would have (some sort of) answer right away. But how do you come to that decision? Is it based on what you ate earlier today, or is it due to chemicals and nutrients your body is deprived of? Our brain can make many complex decisions in a split second, but we do not fully understand how (some of) these decisions are made.

Artificial Neural Networks (ANN) are models from a statistical technique that tries to mimic how our brain makes decision, or at least how neurons work. To many of us, they feel like a black-box methodo (like Figure 1, where you get some stimuli (input), and then an action (output) is taken, but it's not entirely clear what happens in between). In this article, we will investigate what goes on behind the scenes of this black-box technique.

2 ANN in a Nutshell

Mathematically, ANN simply performs a chain of multivariate non-linear regression functions. A trained ANN is a **function** that maps inputs to outputs in a useful way by:

1. receiving input(s);
2. computing values, and
3. providing output(s).

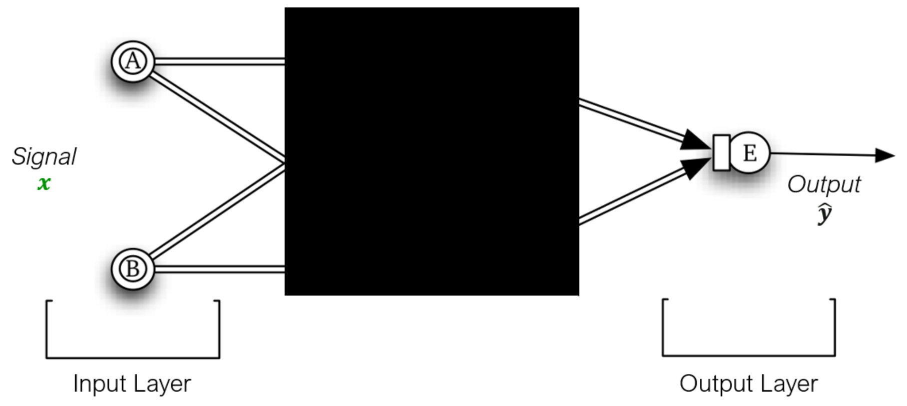

Figure 1. Artificial Neural Networks – a black-box technique?

We can think of an ANN as a Swiss army knife approach to problems (i.e., there are plenty of options, but it is not always clear which one should be used). The good news is, the user does not need to decide much about the function or know much about the problem space in advance – algorithms allow ANNs to learn (i.e., generate the function and its internal values) automatically.

ANNs have many potential uses, including:

1. supervised learning;
2. unsupervised learning, and
3. reinforcement learning.

From a technical perspective, the only requirement is the ability to minimize a cost function (i.e., optimization).

3 Network Topology, Terminology, and Forward Propagation - How Do We Compute the Outputs?

Here, we will describe what is called a *vanilla* neural net, which is also known as *single hidden-layer back propagation network*.

Suppose that we want to build a model that classifies red wines that come from two different regions of Italy, based on two attributes: their colour intensity, and their total amounts of phenols.

In Figure 2, colour intensity and amount of total phenols are the **inputs**, represented by nodes **A** and **B**, while the probability of it coming from a region is the **output**, denoted by node **E**. The nodes **C** and **D** in the hidden layer add complexity (and flexibility) to the model by allowing for non-linear functions to be applied.

In ANN, each node is connected to the nodes in the adjacent layers. This connection, called the **edge**, applies a weight to a signal that passes through it. The receiving node then collects all

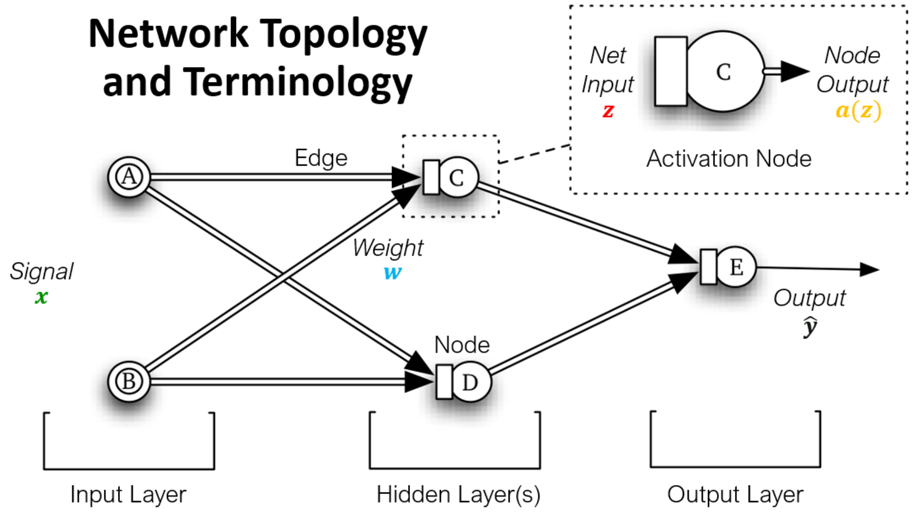

Figure 2. A simple network – illustration

weighted inputs and sum them, apply an activation function, and send its signal to nodes in the subsequent layer. In matrix notation, each component of the above process is summarized as follows:

Input to Hidden Layer

Input: $\boldsymbol{X}_{(n\times p)} = \boldsymbol{X}_{(n\times 2)}$

$$\boldsymbol{X} = \begin{bmatrix} x_{A,1} & x_{B,1} \\ \vdots & \vdots \\ x_{A,n} & x_{B,n} \end{bmatrix}$$

Hidden units: $\boldsymbol{Z}^{(2)}_{(n\times M)} = \boldsymbol{Z}^{(2)}_{(n\times 2)}$

$$\boldsymbol{Z}^{(2)} = \begin{bmatrix} z_{C,1} & z_{D,1} \\ \vdots & \vdots \\ z_{C,n} & z_{D,n} \end{bmatrix} = \boldsymbol{X}\boldsymbol{W}^{(1)}$$

Weights: $\boldsymbol{W}^{(1)}_{(p\times M)} = \boldsymbol{W}^{(1)}_{(2\times 2)}$

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} w_{AC} & w_{AD} \\ w_{BC} & w_{BD} \end{bmatrix}$$

Activation function: $\boldsymbol{a}^{(2)}_{(n\times M)} = \boldsymbol{a}^{(2)}_{(n\times 2)}$

$$\boldsymbol{a}^{(2)} = \begin{bmatrix} 1/\left(1+e^{-(b_C+z_{C,1})}\right) & 1/\left(1+e^{-(b_D+z_{D,1})}\right) \\ \vdots & \vdots \\ 1/\left(1+e^{-(b_C+z_{C,n})}\right) & 1/\left(1+e^{-(b_D+z_{D,n})}\right) \end{bmatrix} = g\left(\boldsymbol{Z}^{(2)}\right)$$

Hidden to Output Layer

Activation function: $\boldsymbol{a}^{(2)}_{(n\times M)} = \boldsymbol{a}^{(2)}_{(n\times 2)}$

$$\boldsymbol{a}^{(2)} = g\left(\boldsymbol{Z}^{(2)}\right)$$

Output units: $\boldsymbol{Z}^{(3)}_{(n\times K)} = \boldsymbol{Z}^{(3)}_{(n\times 1)}$

$$\boldsymbol{Z}^{(3)} = \begin{bmatrix} Z_{E,1} \\ \vdots \\ Z_{E,n} \end{bmatrix} = \boldsymbol{a}^{(2)}\boldsymbol{W}^{(2)}$$

Weights: $\boldsymbol{W}^{(2)}_{(M\times K)} = \boldsymbol{W}^{(2)}_{(2\times 1)}$

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} w_{CE} & w_{DE} \end{bmatrix}$$

Activation function: $\boldsymbol{a}^{(2)}_{(n\times K)} = \boldsymbol{a}^{(2)}_{(n\times 1)}$

$$\hat{\boldsymbol{y}} = \boldsymbol{a}^{(3)} = \begin{bmatrix} 1/\left(1+e^{-(b_E+Z_{E,1})}\right) \\ \vdots \\ 1/\left(1+e^{-(b_E+Z_{E,n})}\right) \end{bmatrix} = g\left(\boldsymbol{Z}^{(3)}\right)$$

where b_C, b_D, and b_E in the activation functions are **biases**, which serve as intercepts for Z. Hence, this *vanilla* neural net can be expressed as

$$\hat{\boldsymbol{y}} = \boldsymbol{a}^{(3)} = g[\boldsymbol{Z}^{(3)}] = g[\boldsymbol{a}^{(2)}\boldsymbol{W}^{(2)}] = g[g(\boldsymbol{X}\boldsymbol{W}^{(1)})\boldsymbol{W}^{(2)}]$$

In short, at each node, the neural net:

1. computes a weighted sum of inputs;
2. applies activation function, and
3. sends a signal,

until the signal reaches the final output node. The process of computing the output is called **forward propagation**.

4 Back Propagation - How Do We Train Our Model?

For a given signal (i.e., data), an ANN can produce an output as long as the weights are specified. However, if we want to use it for supervised learning tasks, assigning arbitrary weights is a failing proposition. To have a useful model, we need a method to optimize the choice of the weights against an error function using **backpropagation**. In regression problems (i.e., when the output is a continuous variable), the **sum of squared errors** (SSE) is often used as the error function.

$$R(\boldsymbol{W}) = \sum_{i=1}^{n}\sum_{k=1}^{K}(\hat{y}_{ik}(\boldsymbol{W}) - y_{ik})^2$$

and if our problem is classification, then **cross-entropy** may be used instead:

$$R(\boldsymbol{W}) = -\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\ln\left[\hat{y}_{ik}(\boldsymbol{W})\right]$$

Either way, our goal is to minimize $R(\boldsymbol{W})$ with respect to $\boldsymbol{W}$. One way to achieve this is through the use of **gradient descent**. Using the **chain rule**, for each weight and bias term we have

$$\begin{aligned} W^{(1)}_{j,m}; j = 1, \cdots, p \text{ and } m = 1, \cdots, M \\ W^{(2)}_{m,k}; m = 1, \cdots, M \text{ and } k = 1, \cdots, K \\ b^{(1)}_{m} \text{ and } b^{(2)}_{k}; m = 1, \cdots, M \text{ and } k = 1, \cdots, K \end{aligned}$$

We update these weights and biases by

$$\begin{aligned} W^{(1),[t+1]}_{j,m} &= W^{(1),[t]}_{j,m} - \alpha_{[t]}\sum_{i=1}^{n}\frac{\partial R_i}{\partial W^{(1),[t]}_{j,m}} \\ W^{(2),[t+1]}_{m,k} &= W^{(2),[t]}_{m,k} - \alpha_{[t]}\sum_{i=1}^{n}\frac{\partial R_i}{\partial W^{(2),[t]}_{m,k}} \\ b^{(1),[t+1]}_{m} &= b^{(1),[t]}_{m} - \alpha_{[t]}\sum_{i=1}^{n}\frac{\partial R_i}{\partial b^{(1),[t]}_{m}} \\ b^{(2),[t+1]}_{k} &= b^{(1),[t]}_{k} - \alpha_{[t]}\sum_{i=1}^{n}\frac{\partial R_i}{\partial b^{(2),[t]}_{k}} \end{aligned}$$

where $\alpha_{[t]}$ is called the **learning rate**, which controls how much adjustment we want to make at each update. It is important to note that if we set $\alpha_{[t]}$ to be too small, the rate of convergence may be very slow, while having $\alpha_{[t]}$ too large may overshoot at each update and miss the local minima completely. Also, as the subscript $[t]$ suggests, the learning rate could be defined to decay over time, or stays a constant.

In essence, backpropagation:

1. computes current error function $R(\boldsymbol{W})$;
2. updates weights $\boldsymbol{W}$ using gradient descent, and
3. recomputes error function and repeat steps 1 and 2 until convergence (or maximum iteration is reached).

5 Pros, Cons, and Other Things to Consider

There certainly is a good reason why many people use ANN to solve problems. Simply said, ANNs can be quite **accurate** when making predictions, often better than other algorithms with a proper set up. Real life data can be ugly and messy, and ANNs often work even when:

- the relationship between attributes is **complex**;
- there are a lot of dependencies/non-linear relationships;
- inputs are highly connected (e.g., images, texts, and speeches), and
- problem is non-linear classification.

On top of these, ANNs are relatively easy to set up using available packages such as *neuralnet* in **R**.

Of course, ANNs are not without their drawbacks. One of their major weakness is the **lack of interpretation** (unlike decision trees or logistic regression, say). Sometimes, understanding the underlying relationship is as important as making good predictions. While ANNs often provides good predictions, the underlying complexity prevents us from understanding how change in input(s) really affect the output.

Overfitting may become an issue with ANNs as well. Remember that we don't want a perfect fit to our data, because such a model will be overly complex and will have poor predictive power. This is especially true when we have too many weights. **Weight decay** is a method that can be used to counter overfitting.

The choice of activation function is another thing the user should think about. We illustrated it previously using the logistic function; however, there are a number of other choices. We could use a a linear function, a threshold function, a hinge function, or even a sine function! While using monotonic functions may seem like an intuitive option, what effects does the choice of activation function have, if any? Users should also be aware that ANNs are sensitive to initial values. Thus, they must be run multiple times to ensure that the method has reached consistent convergence.

Last but not least, what about the number of hidden layers and nodes? Figures 4 and 5 show many different network structures. While we cannot say which architecture works best for a particular problem, we can't tell that just because a structure worked in one situation, it will work in all other situations. What we should remember is that, like any other modeling techniques, we want a model that explains the data well, while keeping it as simple as possible. We will investigate the effect of changing the number of layers and nodes in next section.

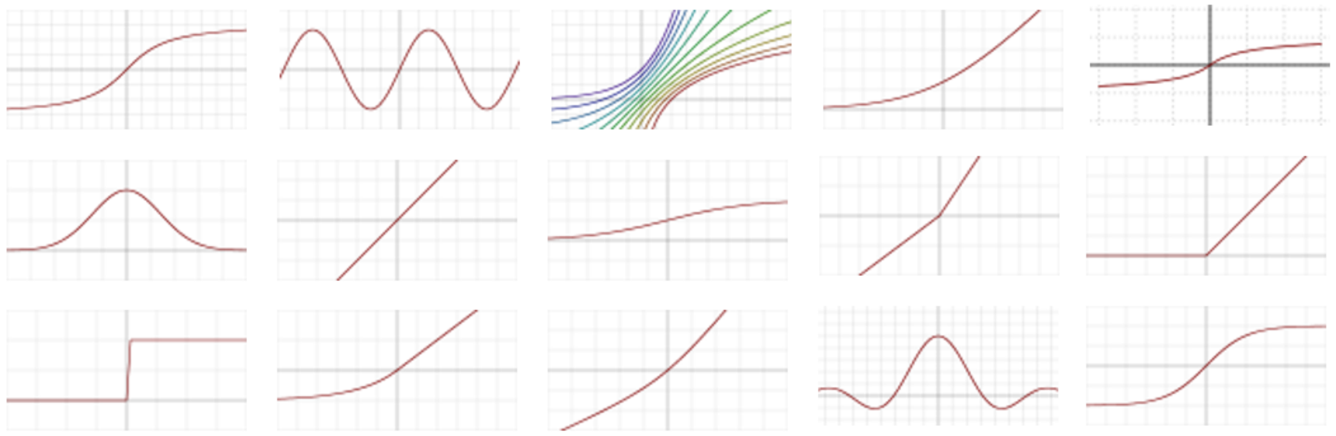
Figure 3. https://en.wikipedia.org/wiki/Activation_function

6 Example: Classification of Wine Using ANN

To illustrate the effect of changing the number of nodes in each hidden layer, let us take a look at the **wine dataset**. Our goal is to build a "good" ANN model that can correctly identify wines that come from three different regions of Italy. The original data contains 178 observations with 13 explanatory variables (alcohol level and amount of malic acid etc.). We use 140 observations to build models, and compare their performance using remaining 38 observations.

Now, let us apply principal component analysis to the standardized data with variable *flavanoids* removed. Figure 6 shows the data projected onto the first two principal components. To our eyes, the classification seem fairly straightforward: red and blue wines can be well separated by a vertical line around 0, while green ones typically have lower PC2 values.

We use following eight ANN models for comparison:

- no hidden layers
- 1 hidden layer with 2 nodes
- 1 hidden layer with 6 nodes
- 1 hidden layer with 10 nodes
- 2 hidden layers with 2 nodes each
- 2 hidden layers with 6 nodes each
- 2 hidden layers with 10 nodes each
- 3 hidden layers with 10 nodes each

and the resulting prediction regions are given in Figures 7 and 8. Interestingly, the two simplest models (i.e., model without hidden layers and model with one hidden layer with two nodes) performed the best, while the prediction regions from more complicated models are clearly overfitted to the training data. Thus, it is recommended to have a simple initial ANN model, and then add complexity as required.

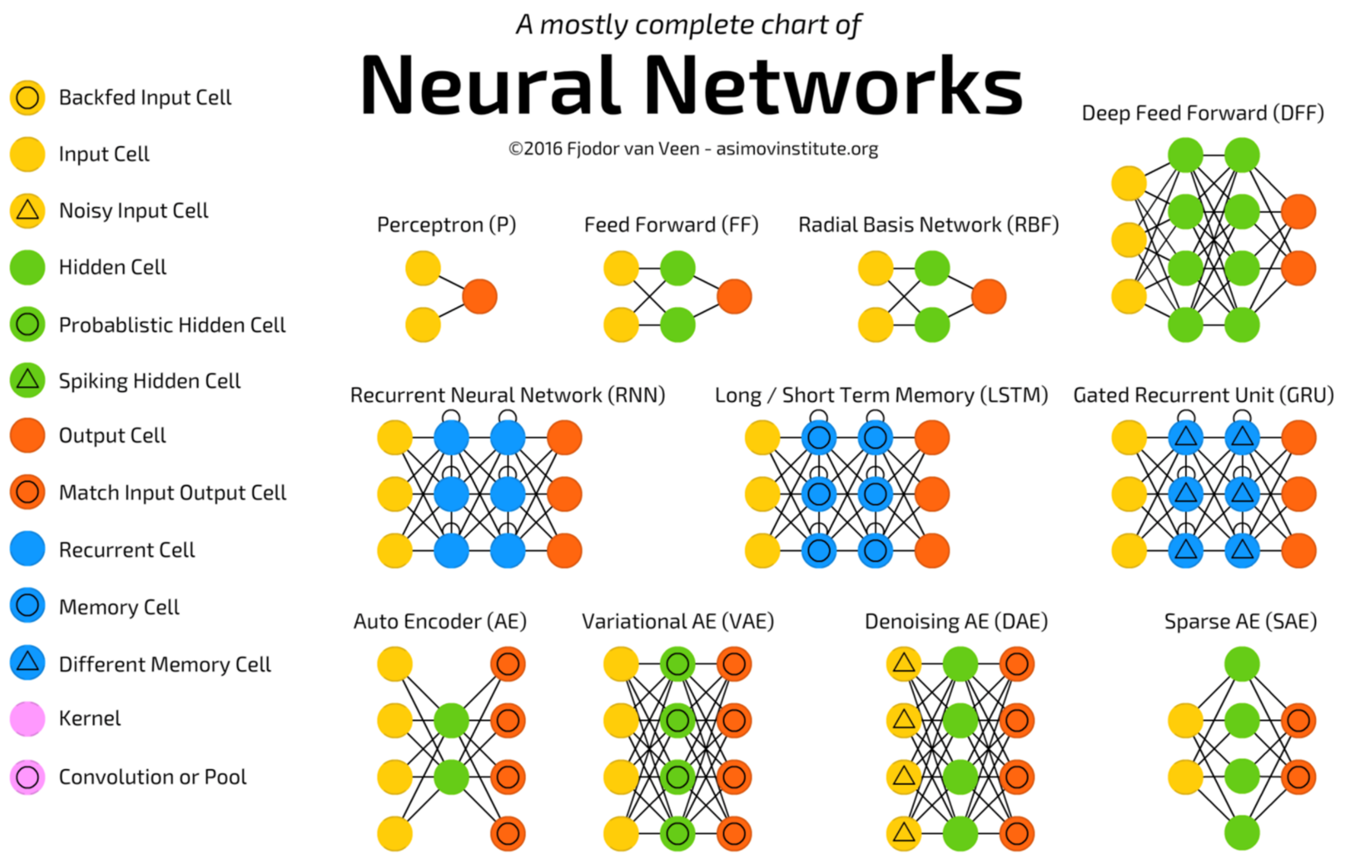

Figure 4. Fjodor van Veen, **Asimov Institute**, 2016

References

[1] Hastie, J., Tibshirani, R., Friedman, J., [2009], *The Elements of Statistical Learning*, 2nd ed., Springer.

[2] Zulkifli, H., **Towards Data Science**

[3] Wikipedia, **https://en.wikipedia.org/wiki/Activation_function**.

[4] van Veen, F., [2016], **Asimov Institute**

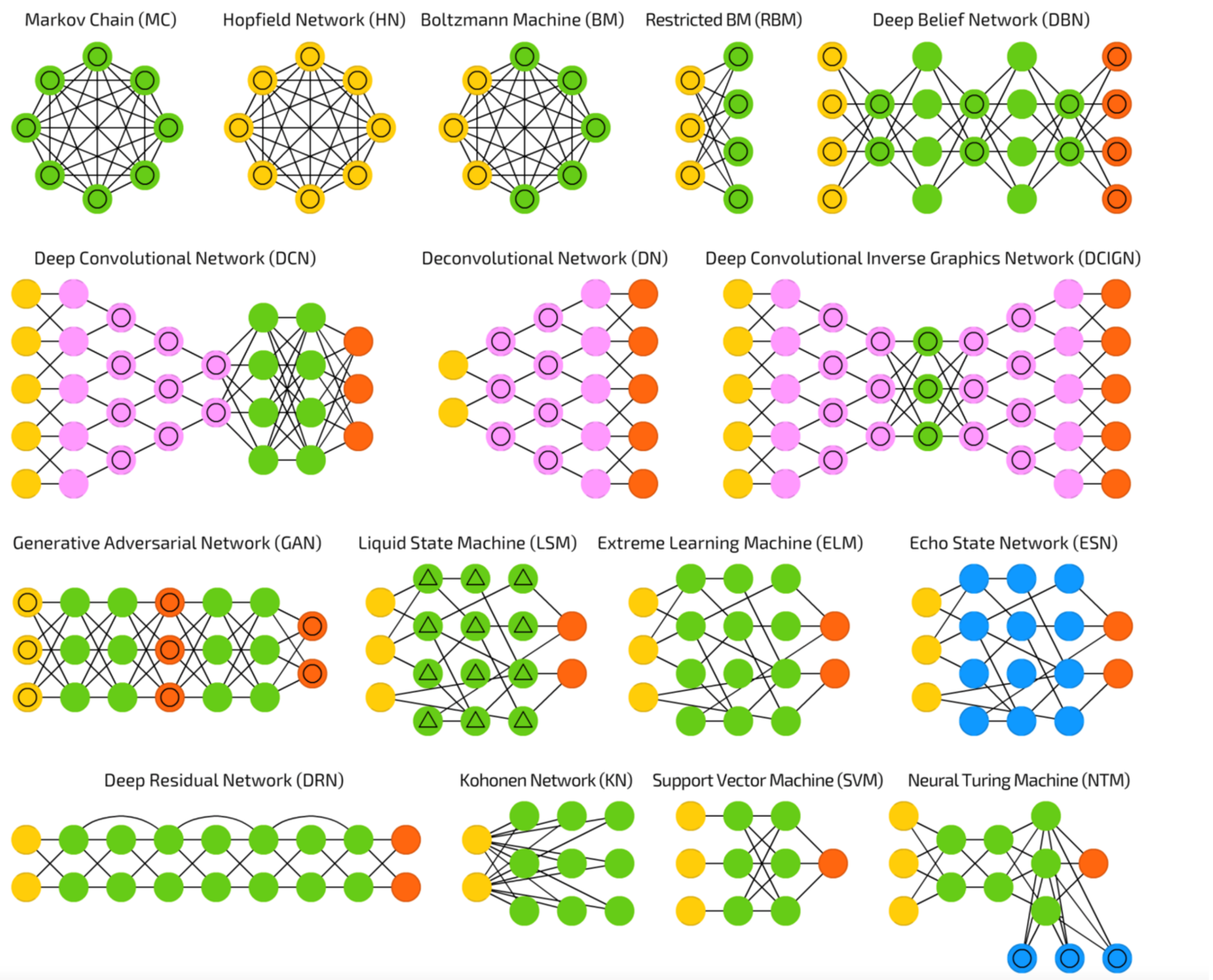
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

Figure 5. Fjodor van Veen, **Asimov Institute**, 2016

7 Appendix A: R Code

```
# Implementing libraries
library(mda)
library(nnet) # required for function class.ind()
library(neuralnet)
library(repr)

# Setting up the data
wine=read.csv("wine.csv", header=TRUE)
wine=as.data.frame(wine)
#str(wine)

n=dim(wine)[1] # Number of instances
```

```
Y=wine$Class # Dependent variable - classes
X=wine[,-1] # Independent variables (full)
X.std=scale(X) # Standardized independent variables (full)
C1.loc=which(Y==1) # Indecies for class=1
C2.loc=which(Y==2) # Indecies for class=2
C3.loc=which(Y==3) # Indecies for class=3

# Remove flavanoid
X=X[,-7]
X.std=X.std[,-7]

# Performing principal component analysis on standardized data
pca.std=prcomp(X.std)

 # Transforming X.std into principal coordinates
PC=X.std%*%pca.std$rotation
PCnames=c("PC1","PC2","PC3","PC4","PC5","PC6","PC7","PC8","PC9","PC10","PC11","PC12")
colnames(PC) <- PCnames

# Splitting dataset into training and testing sets
set.seed(1111)
C1.train.loc=sort(sample(C1.loc, size=46))
C2.train.loc=sort(sample(C2.loc, size=56))
C3.train.loc=sort(sample(C3.loc, size=38))
train.loc=c(C1.train.loc, C2.train.loc, C3.train.loc)
test.loc=which(!(1:length(Y) %in% train.loc))

# Forming training data
PC.train=PC[train.loc,]
Y.train=Y[train.loc]
dat.train=as.data.frame(cbind(class.ind(Y.train), PC.train))
colnames(dat.train)[1:3]=c("C1", "C2", "C3")

# Forming testing data
PC.test=PC[test.loc,]
Y.test=Y[test.loc]
dat.test=as.data.frame(cbind(class.ind(Y.test), PC.test))
colnames(dat.test)[1:3]=c("C1", "C2", "C3")

# Plotting both training (cicles) and testing data (triangles) on PC1 and PC2
plot.title="Training␣and␣Testing␣data"
xlimit=c(-4,4); ylimit=c(-3,3)
plot(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1, main=plot.title, xlab="PC1",
    ylab="PC2", xlim=xlimit, ylim=ylimit)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)
```

```
legend.main=c("Training␣Data", "Testing␣Data")
legend("bottomright", pch=c(1, 17), legend=legend.main)

  #---------------------------------------------------#
  # Artificial Neural Network with neuralnet (part 1) #
  #---------------------------------------------------#

set.seed(1234)
# Form a grid to colour prediction regions
predict.region.PC1=seq(-5,5, length.out=100)
predict.region.PC2=seq(-4,4, length.out=100)
predict.region=expand.grid(x=predict.region.PC1, y=predict.region.PC2)

# Using function neuralnet() from library(neuralnet)

# The number of hidden nodes in each hidden layer
model.structure=0

# Fit an ANN using first 2 principal components only
model1 <- neuralnet(C1+C2+C3~PC1+PC2, data=dat.train, hidden=model.structure, err.fct="
    ce", linear.output=FALSE)
prob.model1 <- compute(model1, PC.train[,1:2])
predict.model1=max.col(prob.model1$net.result)
conf.train=confusion(predict.model1, Y.train)
prob.model1.test <- compute(model1, PC.test[,1:2])
predict.model1.test=max.col(prob.model1.test$net.result)

# Calculate the prediction region based on a particular model (for model1):
prob.model1.region <- compute(model1, predict.region[,1:2])
predict.model1.region=max.col(prob.model1.region$net.result)

# and plot it
options(repr.plot.width=12, repr.plot.height=12)

par(mfrow=c(2,2))
plot.title=paste("Prediction␣region␣for", "\n", "ANN␣with␣structure␣=␣", list(model.
    structure)[1], sep="")
plot(predict.region[,1], predict.region[,2], main=plot.title, xlim=xlimit, ylim=ylimit,
    xlab="PC1", ylab="PC2", col=predict.model1.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)

# The number of hidden nodes in each hidden layer
model.structure=2
```

```
# Fit an ANN using first 2 principal components only
model1 <- neuralnet(C1+C2+C3~PC1+PC2, data=dat.train, hidden=model.structure, err.fct="
    ce", linear.output=FALSE)
prob.model1 <- compute(model1, PC.train[,1:2])
predict.model1=max.col(prob.model1$net.result)
conf.train=confusion(predict.model1, Y.train)
prob.model1.test <- compute(model1, PC.test[,1:2])
predict.model1.test=max.col(prob.model1.test$net.result)

# Calculate the prediction region based on a particular model (for model1):
prob.model1.region <- compute(model1, predict.region[,1:2])
predict.model1.region=max.col(prob.model1.region$net.result)

# and plot it
plot.title=paste("Prediction␣region␣for", "\n", "ANN␣with␣structure␣=␣", list(model.
    structure)[1], sep="")
plot(predict.region[,1], predict.region[,2], main=plot.title, xlim=xlimit, ylim=ylimit,
    xlab="PC1", ylab="PC2", col=predict.model1.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)

# The number of hidden nodes in each hidden layer
model.structure=6

# Fit an ANN using first 2 principal components only
model1 <- neuralnet(C1+C2+C3~PC1+PC2, data=dat.train, hidden=model.structure, err.fct="
    ce", linear.output=FALSE)
prob.model1 <- compute(model1, PC.train[,1:2])
predict.model1=max.col(prob.model1$net.result)
conf.train=confusion(predict.model1, Y.train)
prob.model1.test <- compute(model1, PC.test[,1:2])
predict.model1.test=max.col(prob.model1.test$net.result)

# Calculate the prediction region based on a particular model (for model1):
prob.model1.region <- compute(model1, predict.region[,1:2])
predict.model1.region=max.col(prob.model1.region$net.result)

# and plot it
plot.title=paste("Prediction␣region␣for", "\n", "ANN␣with␣structure␣=␣", list(model.
    structure)[1], sep="")
plot(predict.region[,1], predict.region[,2], main=plot.title, xlim=xlimit, ylim=ylimit,
    xlab="PC1", ylab="PC2", col=predict.model1.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)

# The number of hidden nodes in each hidden layer
```

```
model.structure=10

# Fit an ANN using first 2 principal components only
model1 <- neuralnet(C1+C2+C3~PC1+PC2, data=dat.train, hidden=model.structure, err.fct="
    ce", linear.output=FALSE)
prob.model1 <- compute(model1, PC.train[,1:2])
predict.model1=max.col(prob.model1$net.result)
conf.train=confusion(predict.model1, Y.train)
prob.model1.test <- compute(model1, PC.test[,1:2])
predict.model1.test=max.col(prob.model1.test$net.result)

# Calculate the prediction region based on a particular model (for model1):
prob.model1.region <- compute(model1, predict.region[,1:2])
predict.model1.region=max.col(prob.model1.region$net.result)

# and plot it
plot.title=paste("Prediction␣region␣for", "\n", "ANN␣with␣structure␣=␣", list(model.
    structure)[1], sep="")
plot(predict.region[,1], predict.region[,2], main=plot.title, xlim=xlimit, ylim=ylimit,
    xlab="PC1", ylab="PC2", col=predict.model1.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)

  #----------------------------------------------------#
  # Artificial Neural Network with neuralnet (part 2) #
  #----------------------------------------------------#

set.seed(1234)
# Form a grid to colour prediction regions
predict.region.PC1=seq(-5,5, length.out=100)
predict.region.PC2=seq(-4,4, length.out=100)
predict.region=expand.grid(x=predict.region.PC1, y=predict.region.PC2)

# Using function neuralnet() from library(neuralnet)

# The number of hidden nodes in each hidden layer
model.structure=c(2,2)

# Fit an ANN using first 2 principal components only
model1 <- neuralnet(C1+C2+C3~PC1+PC2, data=dat.train, hidden=model.structure, err.fct="
    ce", linear.output=FALSE)
prob.model1 <- compute(model1, PC.train[,1:2])
predict.model1=max.col(prob.model1$net.result)
conf.train=confusion(predict.model1, Y.train)
prob.model1.test <- compute(model1, PC.test[,1:2])
predict.model1.test=max.col(prob.model1.test$net.result)
```

```
# Calculate the prediction region based on a particular model
# For model1:
prob.model1.region <- compute(model1, predict.region[,1:2])
predict.model1.region=max.col(prob.model1.region$net.result)

# and plot it
options(repr.plot.width=12, repr.plot.height=12)

par(mfrow=c(2,2))
plot.title=paste("Prediction␣region␣for", "\n", "ANN␣with␣structure␣=␣", list(model.
    structure)[1], sep="")
plot(predict.region[,1], predict.region[,2], main=plot.title, xlim=xlimit, ylim=ylimit,
    xlab="PC1", ylab="PC2", col=predict.model1.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)

# The number of hidden nodes in each hidden layer
model.structure=c(6,6)

# Fit an ANN using first 2 principal components only
model1 <- neuralnet(C1+C2+C3~PC1+PC2, data=dat.train, hidden=model.structure, err.fct="
    ce", linear.output=FALSE)
prob.model1 <- compute(model1, PC.train[,1:2])
predict.model1=max.col(prob.model1$net.result)
conf.train=confusion(predict.model1, Y.train)
prob.model1.test <- compute(model1, PC.test[,1:2])
predict.model1.test=max.col(prob.model1.test$net.result)

# Calculate the prediction region based on a particular model
# For model1:
prob.model1.region <- compute(model1, predict.region[,1:2])
predict.model1.region=max.col(prob.model1.region$net.result)

# and plot it
plot.title=paste("Prediction␣region␣for", "\n", "ANN␣with␣structure␣=␣", list(model.
    structure)[1], sep="")
plot(predict.region[,1], predict.region[,2], main=plot.title, xlim=xlimit, ylim=ylimit,
    xlab="PC1", ylab="PC2", col=predict.model1.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)

# The number of hidden nodes in each hidden layer
model.structure=c(10,10)

# Fit an ANN using first 2 principal components only
```

```
model1 <- neuralnet(C1+C2+C3~PC1+PC2, data=dat.train, hidden=model.structure, err.fct="
    ce", linear.output=FALSE)
prob.model1 <- compute(model1, PC.train[,1:2])
predict.model1=max.col(prob.model1$net.result)
conf.train=confusion(predict.model1, Y.train)
prob.model1.test <- compute(model1, PC.test[,1:2])
predict.model1.test=max.col(prob.model1.test$net.result)

# Calculate the prediction region based on a particular model
# For model1:
prob.model1.region <- compute(model1, predict.region[,1:2])
predict.model1.region=max.col(prob.model1.region$net.result)

# and plot it
plot.title=paste("Prediction␣region␣for", "\n", "ANN␣with␣structure␣=␣", list(model.
    structure)[1], sep="")
plot(predict.region[,1], predict.region[,2], main=plot.title, xlim=xlimit, ylim=ylimit,
    xlab="PC1", ylab="PC2", col=predict.model1.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)

# The number of hidden nodes in each hidden layer
model.structure=c(10,10,10)

# Fit an ANN using first 2 principal components only
model1 <- neuralnet(C1+C2+C3~PC1+PC2, data=dat.train, hidden=model.structure, err.fct="
    ce", linear.output=FALSE)
prob.model1 <- compute(model1, PC.train[,1:2])
predict.model1=max.col(prob.model1$net.result)
conf.train=confusion(predict.model1, Y.train)
prob.model1.test <- compute(model1, PC.test[,1:2])
predict.model1.test=max.col(prob.model1.test$net.result)

# Calculate the prediction region based on a particular model
# For model1:
prob.model1.region <- compute(model1, predict.region[,1:2])
predict.model1.region=max.col(prob.model1.region$net.result)

# and plot it
plot.title=paste("Prediction␣region␣for", "\n", "ANN␣with␣structure␣=␣", list(model.
    structure)[1], sep="")
plot(predict.region[,1], predict.region[,2], main=plot.title, xlim=xlimit, ylim=ylimit,
    xlab="PC1", ylab="PC2", col=predict.model1.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5, col=Y.test+1)
```

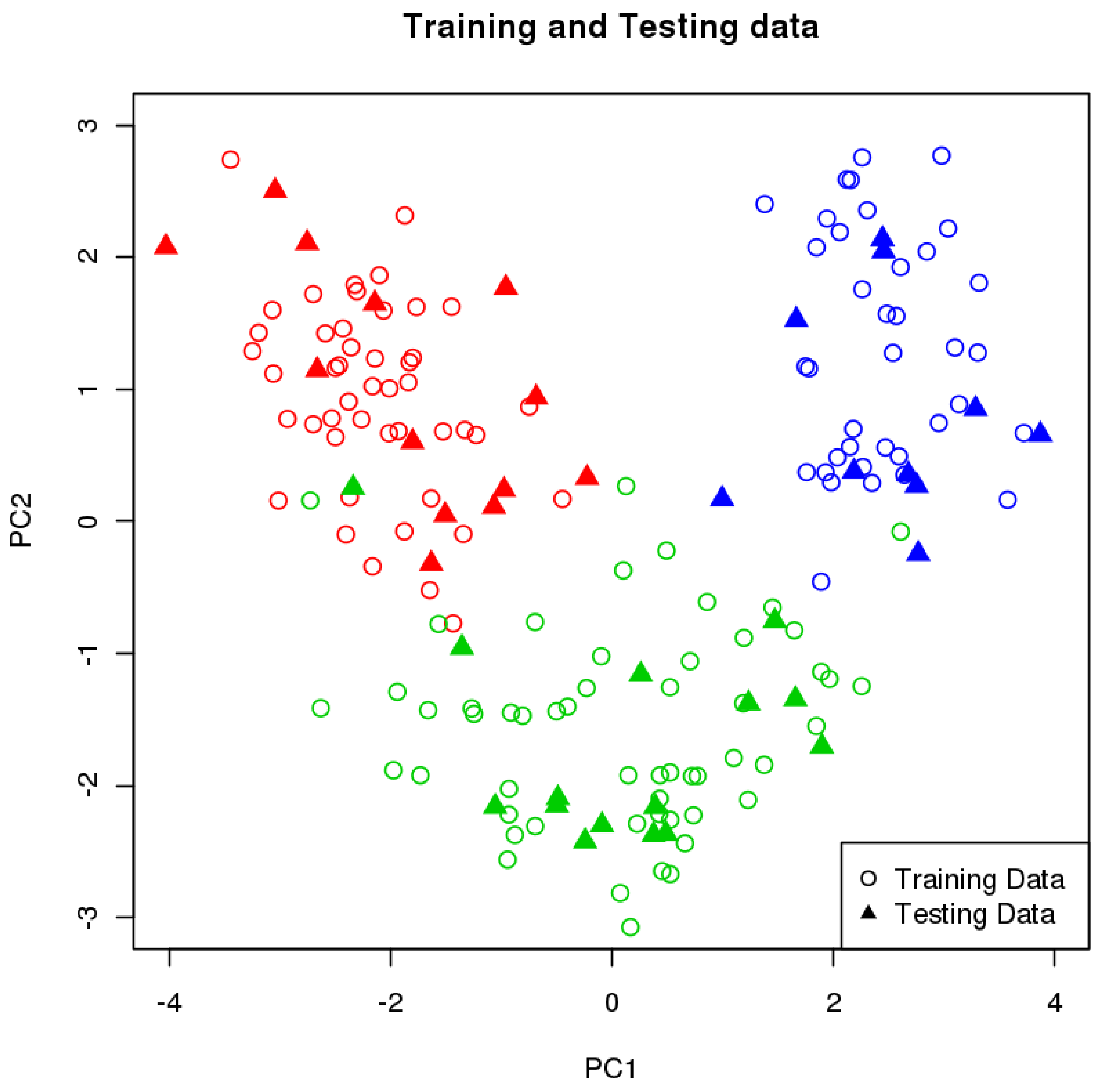

Figure 6. Wine dataset

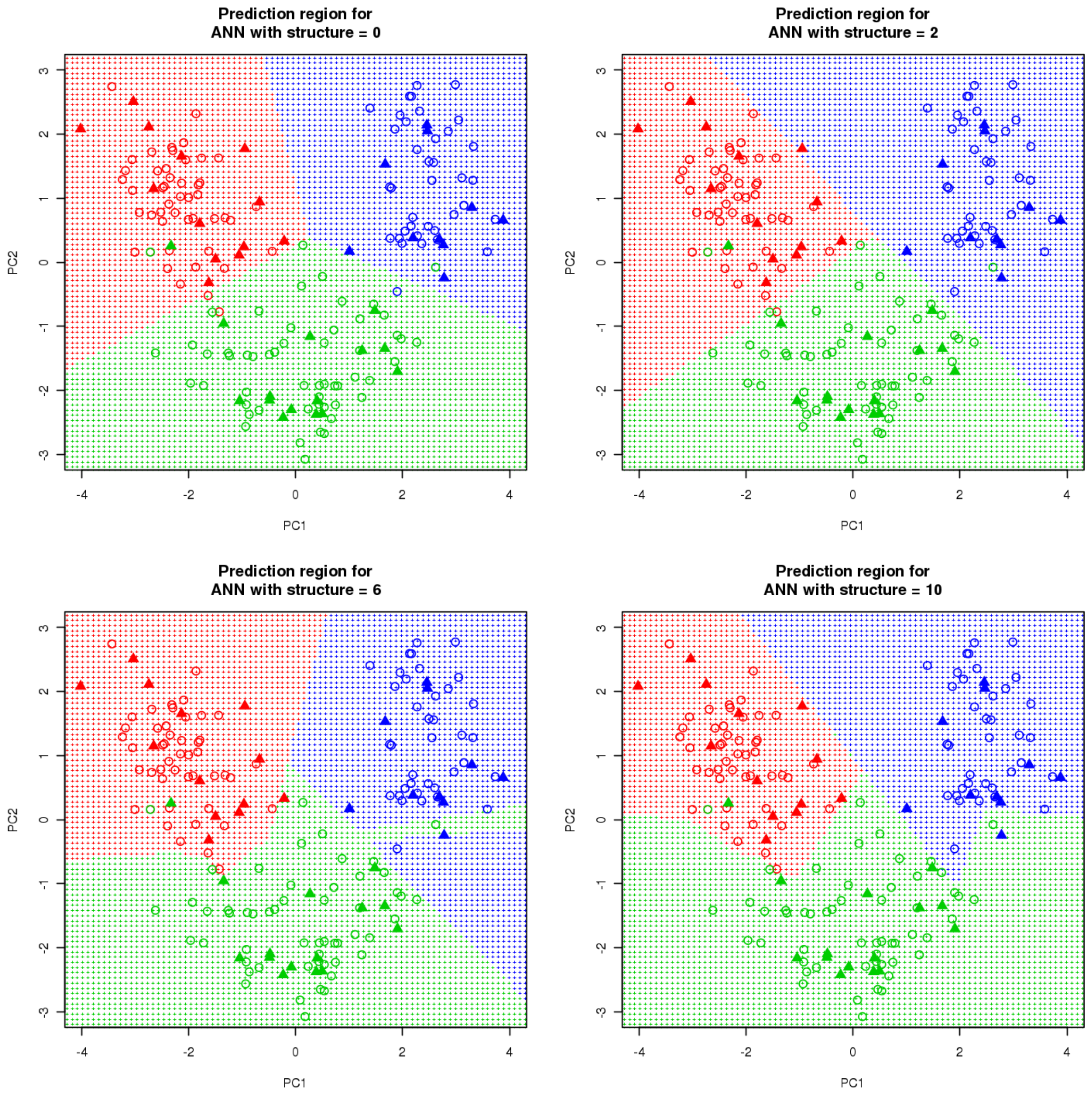

Figure 7. Prediction regions based on different ANNs

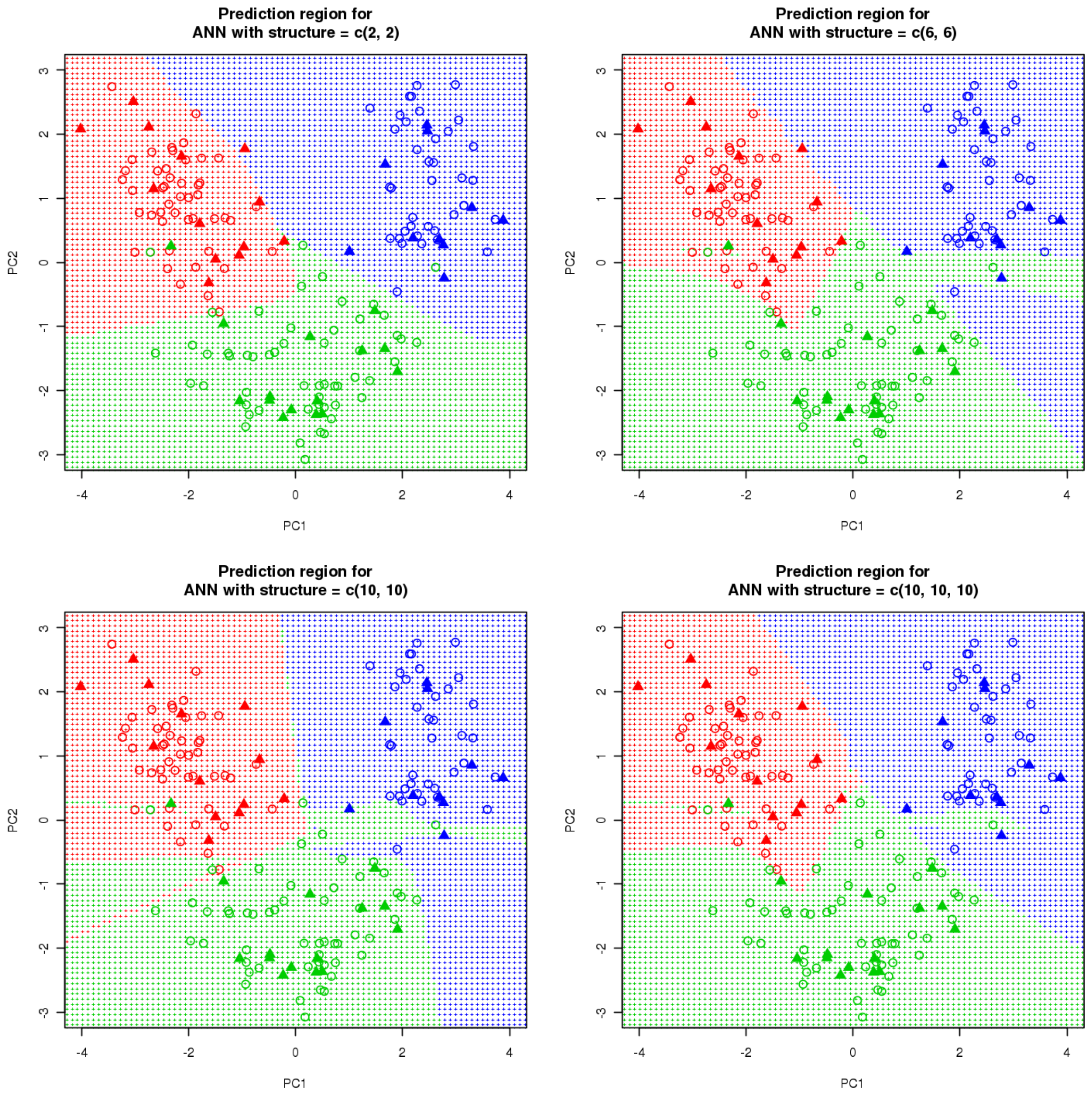

Figure 8. Prediction regions based on different ANNs (continued)