**רשת נוירונים מלאכותית:**

**1.הסבר כללי:**

**הגדרה:**

ANN היא מודל להערכת פונקציות אשר תלויות במספר גבוה של קלטים המבוססת על לימוד איטראטיבי.

בתרגיל זה אנו נלמד ANN לזיהוי וסיווג תמונות רפואיות, מספרים ותמונות כלליות(מורכב מ10 מחלקות: כלבים, צפרדעים, מכוניות ועוד).

**מבנה הרשת:**

ANN מורכבת ממספר רב של קודקודים המחוברים ביניהם בקשתות, כך שכל קודקוד מבצע חישוב פשוט יחסית.

הרשת מאופיינת על ידי:

-חיבורים-אופן החיבור בין הקודקודים ברשת.

-משקלים-השיטה הקובעת את משקלי החיבורים בין הנוירונים.

-פונקציית האקטיבציה: ממירה את הקלט של השכבה אל הפלט שלה.

**הרשת בנויה ממספר שכבות:**

שכבת הקלט: תפקידה לקלוט מידע לרשת

השכבה(ות) החבויות: תפקידיה לעבד את המידע.

שכבת הקלט: תפקידה להוציא את המידע המעובד כפלט.

**אלגוריתם ללימוד הרשת:**

הרשת לומדת לפי אלגוריתם לימוד מסוים(אצלנו באמצעות Back-Propagation) אשר מבצע כוונון של המשקולות לערכים שנותנים תוצאות טובות יותר עבור מערך דוגמאות הלימוד.

**אלגוריתם Back-Propagation:**

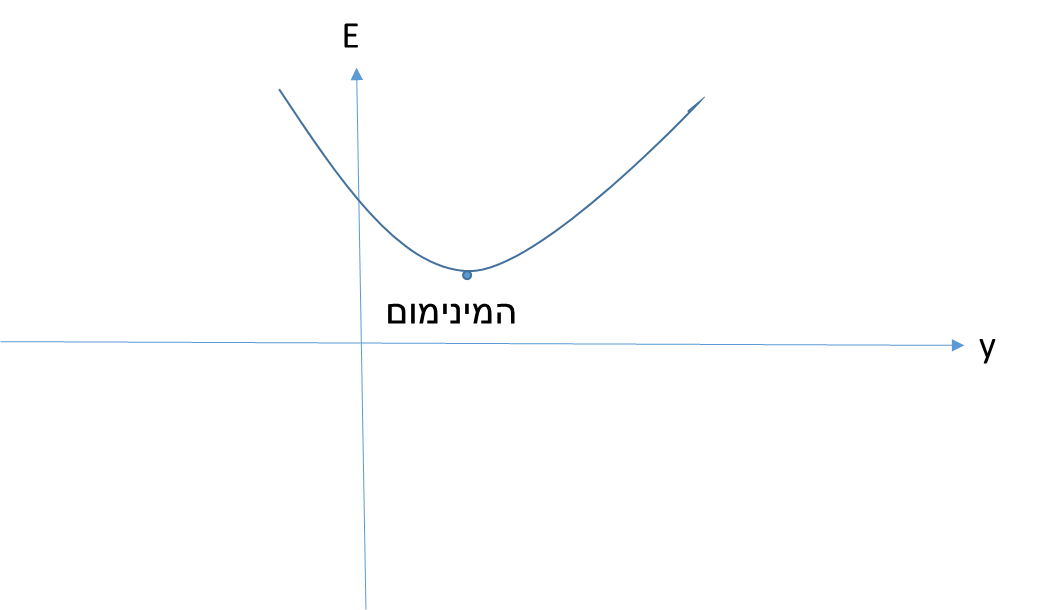
זהו אלגוריתם ללימוד מונחה.

הוא מעביר את הקלט לימוד במורד הרשת עם משקולות אקראיים, משווה את הפלט המתקבל לפלט הרצוי בכך שהוא מחשב את השגיאה עבור כל פלט של נוירון.

שיטה לבדיקת השגיאה בין הפלט המתקבל(y) לפלט הרצוי(t):

E=(t-y)^2.

כלומר בגרף ששני הצירים הן E ו y מתקבלת פרבולה כך שהמינימום של הפרבולה נותן את השגיאה המינימלית.



נקודת המינימום הנ"ל, אשר נותנת את סט המשקולות שמצמצת את השגיאה בין הפלט המתקבל לפלט הרצוי, מחושבת באמצעות שיטת הגרדיאנט.

**שיטת הגרדיאנט:**

האלגוריתם עובד על ידי לקיחת הנגזרת של פונקציית המחיר(שהיא בעצם השגיאה) ביחס למשקולות בנקודה מסוימת על הפונקציה ועדכון הפרמטרים בכיוון השלילי של הגרדיאנט, עד אשר מתקבל (אם מתכנס) המינימום של פונקציית השגיאה.

**2.קטעי קוד מרכזיים:**

**הפונקציות:**

get\_splitted\_data\_for\_Cyst()

get\_splitted\_data\_for\_Cifar()

load\_and\_split\_mnist()

תפקידן לטעון את התמונות לתוך dataset לצורך שימוש שלהן באימון הרשת.

**הפונקציה הלוגיסטית: def logistic(z):**

מספקת את ההסתברות של הקלט z להשתייך למחלקה מסוימת.

*# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# Define the non-linear functions used  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -***def** logistic(z):  
 **return** 1 / (1 + np.exp(-z))  
**def** logistic\_deriv(y): *# Derivative of logistic function* **return** np.multiply(y, (1 - y))

**פונקציית הsoftmax: def softmax(z)**

ממירה תוצאה גולמית להסתברות להיות במחלקה מסוימת.

**def** softmax(z):  
 **return** np.exp(z) / np.sum(np.exp(z), axis=1, keepdims=True)

**המחלקה Layer:class layer**

מחלקה שתפקידה מימוש אבסטרקטי של שכבה ברשת.

*# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# Define the layers used in this model  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -***class** Layer(object):  
 *"""Base class for the different layers.  
 Defines base methods and documentation of methods."""* **def** get\_params\_iter(self):  
 *"""Return an iterator over the parameters (if any).  
 The iterator has the same order as get\_params\_grad.  
 The elements returned by the iterator are editable in-place."""* **return** []  
  
 **def** get\_params\_grad(self, X, output\_grad):  
 *"""Return a list of gradients over the parameters.  
 The list has the same order as the get\_params\_iter iterator.  
 X is the input.  
 output\_grad is the gradient at the output of this layer.  
 """* **return** []  
  
 **def** get\_output(self, X):  
 *"""Perform the forward step linear transformation.  
 X is the input."""* **pass  
  
 def** get\_input\_grad(self, Y, output\_grad=None, T=None):  
 *"""Return the gradient at the inputs of this layer.  
 Y is the pre-computed output of this layer (not needed in this case).  
 output\_grad is the gradient at the output of this layer  
 (gradient at input of next layer).  
 Output layer uses targets T to compute the gradient based on the  
 output error instead of output\_grad"""* **pass**

**המחלקה class LinearLayer:**

מחלקה שתפקידה מימוש אבסטרקטי של שכבה ברשת ליניארית.

*# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# LinearLayer  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -***class** LinearLayer(Layer):  
 *"""The linear layer performs a linear transformation to its input."""* **def** \_\_init\_\_(self, n\_in, n\_out):  
 *"""Initialize hidden layer parameters.  
 n\_in is the number of input variables.  
 n\_out is the number of output variables."""* self.W = np.random.randn(n\_in, n\_out) \* 0.1  
 self.b = np.zeros(n\_out)  
  
 **def** get\_params\_iter(self):  
 *"""Return an iterator over the parameters."""* **return** itertools.chain(np.nditer(self.W, op\_flags=[**'readwrite'**]),  
 np.nditer(self.b, op\_flags=[**'readwrite'**]))  
  
 **def** get\_output(self, X):  
 *"""Perform the forward step linear transformation."""* **return** X.dot(self.W) + self.b  
  
 **def** get\_params\_grad(self, X, output\_grad):  
 *"""Return a list of gradients over the parameters."""* JW = X.T.dot(output\_grad)  
 Jb = np.sum(output\_grad, axis=0)  
 **return** [g **for** g **in** itertools.chain(np.nditer(JW), np.nditer(Jb))]  
  
 **def** get\_input\_grad(self, Y, output\_grad):  
 *"""Return the gradient at the inputs of this layer."""* **return** output\_grad.dot(self.W.T)

**המחלקה class LogisticLayer:**

מחלקה שתפקידה ביצוע פעולות לוגיסטיות על שכבה ברשת.

*# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# LogisticLayer  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -***class** LogisticLayer(Layer):  
 *"""The logistic layer applies the logistic function to its"""* **def** get\_output(self, X):  
 *"""Perform the forward step transformation."""* **return** logistic(X)  
  
 **def** get\_input\_grad(self, Y, output\_grad):  
 *"""Return the gradient at the inputs of this layer."""* **return** np.multiply(logistic\_deriv(Y), output\_grad)

**המחלקה class SoftmaxOutputLayer:**

מחלקה שמממשת את שכבת הsoftmax ברשת ,אשר תפקידה לחשב את ההסתברות להשתייכות למחלקה מסוימת בפלט.

*# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# SoftmaxOutputLayer  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -***class** SoftmaxOutputLayer(Layer):  
 *"""The softmax output layer computes the classification propabilities at the output."""* **def** get\_output(self, X):  
 *"""Perform the forward step transformation."""* **return** softmax(X)  
  
 **def** get\_input\_grad(self, Y, T):  
 *"""Return the gradient at the inputs of this layer."""* **return** (Y - T) / Y.shape[0]  
  
 **def** get\_cost(self, Y, T):  
 *"""Return the cost at the output of this output layer."""* **return** - np.multiply(T, np.log(Y)).sum() / Y.shape[0]

**forward propagation:**

תפקידו להעביר את הקלט בכל הרשת החל משכבת הקלט.

*# forward  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# Define the forward propagation step as a method.***def** forward\_step(input\_samples, layers):  
 *"""  
 Compute and return the forward activation of each layer in layers.  
 Input:  
 input\_samples: A matrix of input samples (each row is an input vector)  
 layers: A list of Layers  
 Output:  
 A list of activations where the activation at each index i+1 corresponds to  
 the activation of layer i in layers. activations[0] contains the input samples.  
 """* activations = [input\_samples] *# List of layer activations  
 # Compute the forward activations for each layer starting from the first* X = input\_samples  
 **for** layer **in** layers:  
 Y = layer.get\_output(X) *# Get the output of the current layer* activations.append(Y) *# Store the output for future processing* X = activations[-1] *# Set the current input as the activations of the previous layer* **return** activations *# Return the activations of each layer*

**Back-Propagation:**

תפקידו להריץ את האלגוריתם Back-Propagation תוך שימוש באלגוריתם שיטת הגרדיאנט ועדכון המשקולות בהתאם.

*# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# Backward step  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# Define the backward propagation step as a method***def** backward\_step(activations, targets, layers):  
 *"""  
 Perform the backpropagation step over all the layers and return the parameter gradients.  
 Input:  
 activations: A list of forward step activations where the activation at  
 each index i+1 corresponds to the activation of layer i in layers.  
 activations[0] contains the input samples.  
 targets: The output targets of the output layer.  
 layers: A list of Layers corresponding that generated the outputs in activations.  
 Output:  
 A list of parameter gradients where the gradients at each index corresponds to  
 the parameters gradients of the layer at the same index in layers.  
 """* param\_grads = collections.deque() *# List of parameter gradients for each layer* output\_grad = None *# The error gradient at the output of the current layer  
 # Propagate the error backwards through all the layers.  
 # Use reversed to iterate backwards over the list of layers.* **for** layer **in** reversed(layers):  
 Y = activations.pop() *# Get the activations of the last layer on the stack  
 # Compute the error at the output layer.  
 # The output layer error is calculated different then hidden layer error.* **if** output\_grad **is** None:  
 input\_grad = layer.get\_input\_grad(Y, targets)  
 **else**: *# output\_grad is not None (layer is not output layer)* input\_grad = layer.get\_input\_grad(Y, output\_grad)  
 *# Get the input of this layer (activations of the previous layer)* X = activations[-1]  
 *# Compute the layer parameter gradients used to update the parameters* grads = layer.get\_params\_grad(X, output\_grad)  
 param\_grads.appendleft(grads)  
 *# Compute gradient at output of previous layer (input of current layer):* output\_grad = input\_grad  
 **return** list(param\_grads) *# Return the parameter gradients  
  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# update\_params  
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  
# Define a method to update the parameters***def** update\_params(layers, param\_grads, learning\_rate):  
 *"""  
 Function to update the parameters of the given layers with the given gradients  
 by gradient descent with the given learning rate.  
 """* **for** layer, layer\_backprop\_grads **in** zip(layers, param\_grads):  
 **for** param, grad **in** itertools.izip(layer.get\_params\_iter(), layer\_backprop\_grads):  
 *# The parameter returned by the iterator point to the memory space of  
 # the original layer and can thus be modified inplace.* param -= learning\_rate \* grad *# Update each parameter*

**בניית הרשת:**

הרשת תהיה מורכבת מ3 שכבות : שכבת קלט, שכבה חבויה אחת ושכבת הפלט.

*# Create the model*layers = [] *# Define a list of layers  
# Add first hidden layer*layers.append(LinearLayer(X\_train.shape[1], hidden\_neurons\_1))  
layers.append(LogisticLayer())  
*# Add second hidden layer*layers.append(LinearLayer(hidden\_neurons\_1, hidden\_neurons\_2))  
layers.append(LogisticLayer())  
*# Add output layer*layers.append(LinearLayer(hidden\_neurons\_2, T\_train.shape[1]))  
layers.append(SoftmaxOutputLayer())

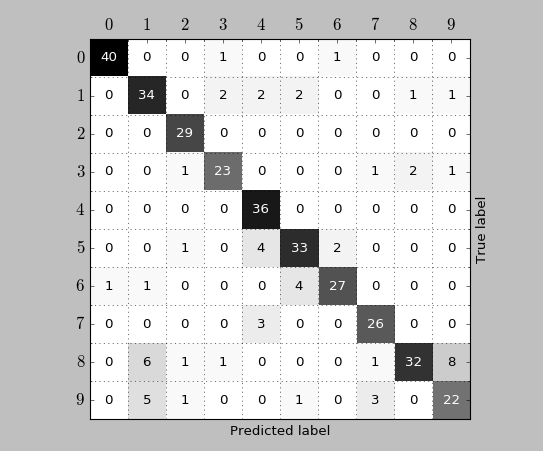
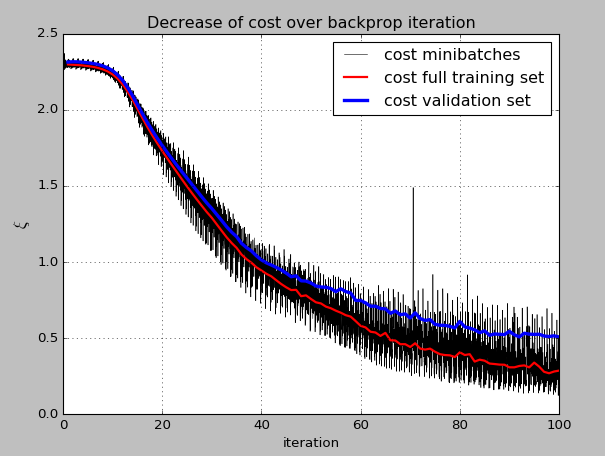
**3.פלט וגרפים:**

**נראה תוצאות שונות בקונפיגורציות שונות על הdataset MNIST, כדי לתאר את ההשפעות השונות של פרמטרים שונים בלימוד הרשת**

**1.מספר נוירונים גבוה בשכבה החבויה מניב תוצאות חישוב מדויקות יותר:**

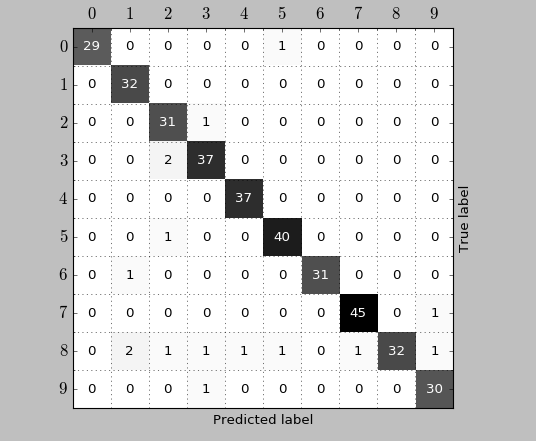
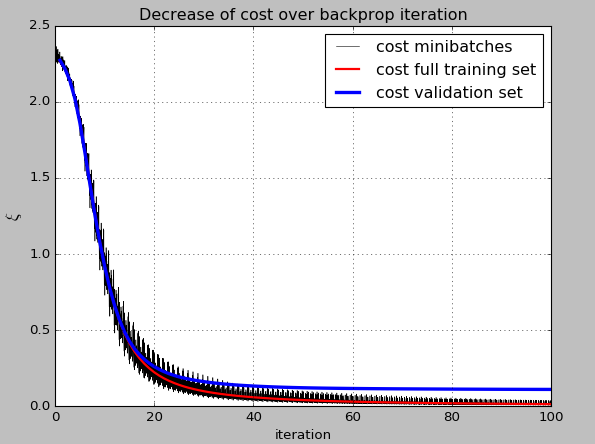
neurons in hidden layer: 10

תוצאות:



neurons in hidden layer: 40

תוצאות:



מסקנות:

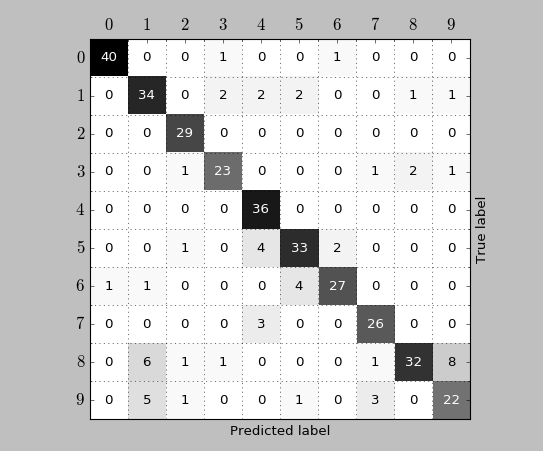
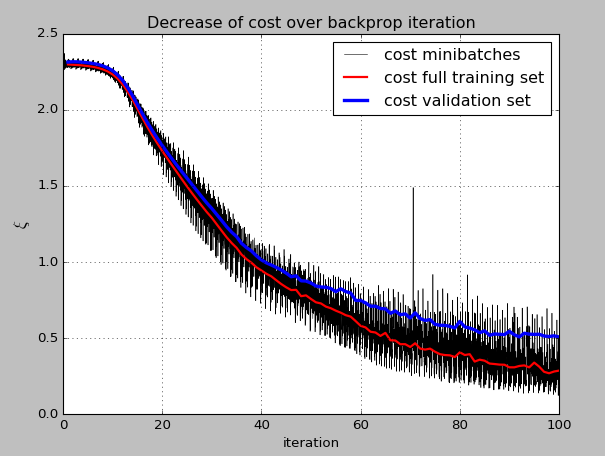
ניתן לראות שפונקציית המחיר יורדת מהר יותר במהלך האיטרציות כאשר יש מספר נוירונים גבוה יותר בשכבה החבויה.

כמו כן, השגיאה מצמצמת משמעותית יותר בסיום האלגוריתם.

**2.קצב לימוד גבוה יותר מניב תוצאות פחות טובות:**

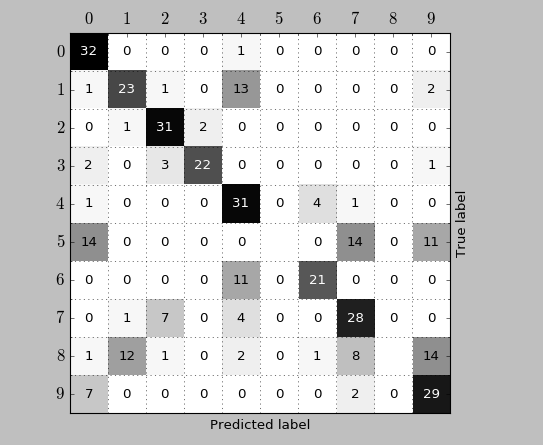
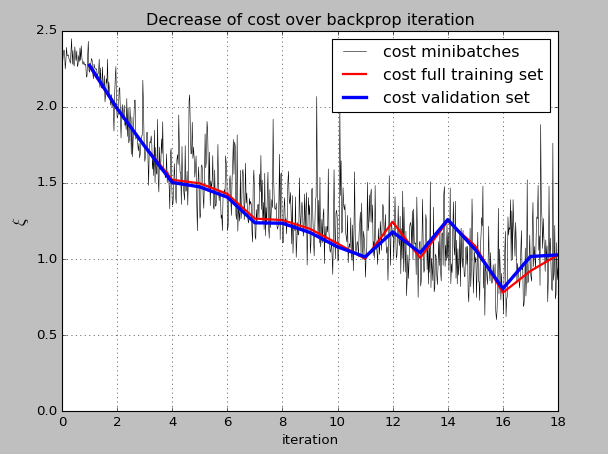
learning rate: 0.1

תוצאות:



learning rate: 0.8

תוצאות:

****

**מסקנות:**

ככל שקצב הלימוד גבוה יותר כך הלימוד פחות טוב והחיזוי פחות טוב.

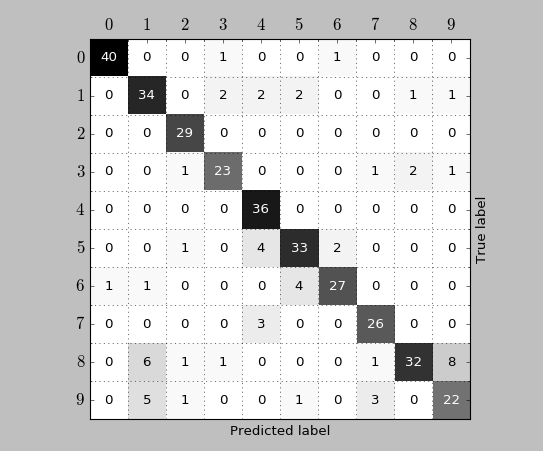
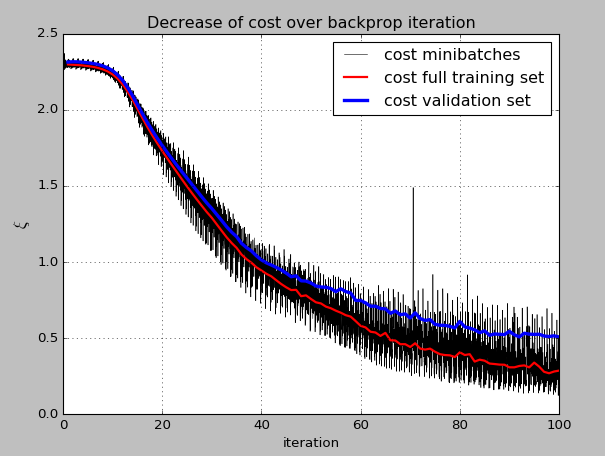
זאת מכיוון שהאלגוריתם משתמש בשיטת הגרדיאנט למציאת נקודת המינימום של פונקציית המחיר:

האלגוריתם מתקדם על הפונקציה בכיוון השלילי של הגרדיאנט בקצב של קצב הלימוד; כתוצאה מכך, ככל שקצב הלימוד קטן יותר כך האלגוריתם מתקדם על הגרדיאנט צעדים יותר קטנים ויותר מדויקים ; דבר אשר מניב תוצאות מדויקות יותר.

3.ככל שמספר האיטרציות קטן יותר , כך הדיוק גבוה יותר:

number of iteration: 100

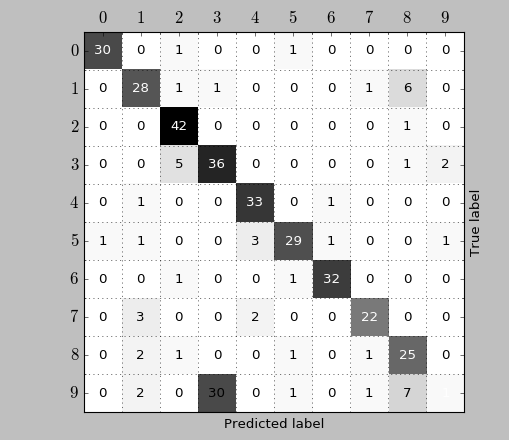
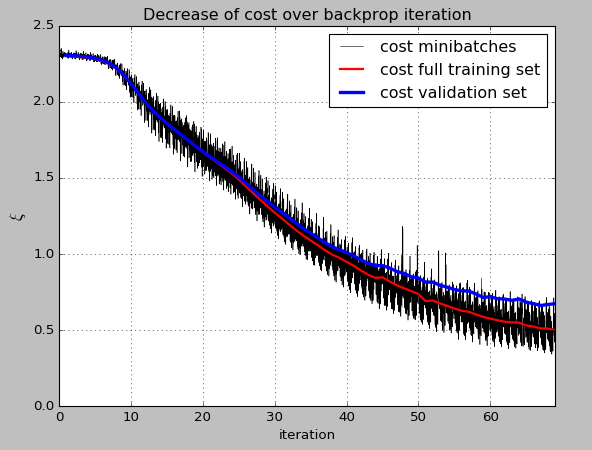
תוצאות:



רמת דיוק: 0.76

number of iteration: 300

תוצאות:



רמת דיוק:0.63

מסקנות:

ניתן לראות שככל שמספר האיטרציות קטן יותר כך רמת הדיוק גבוהה יותר.