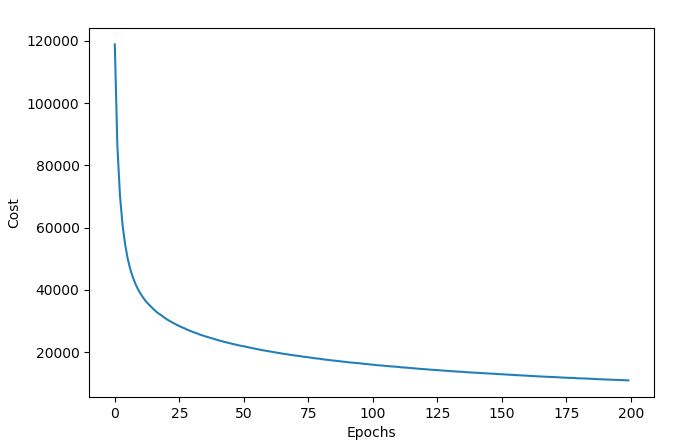
Kyle Calabro

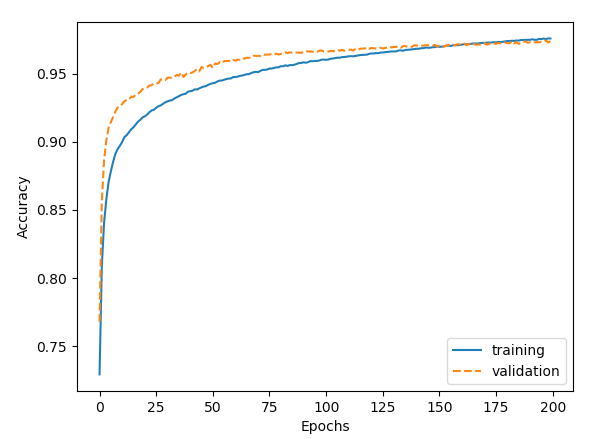
Dr. Kumar

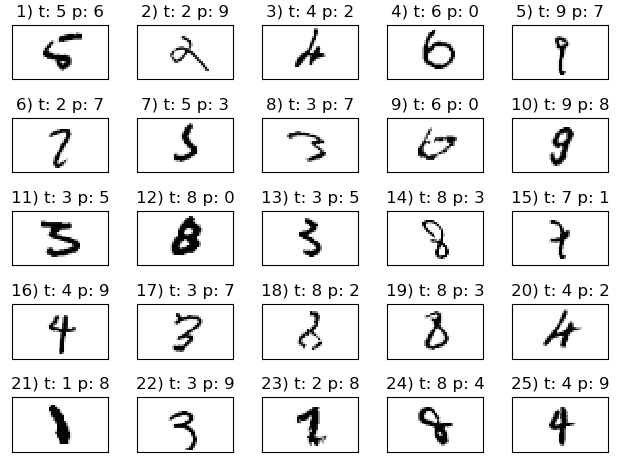
Artificial Intelligence – Project 5

29 April 2018

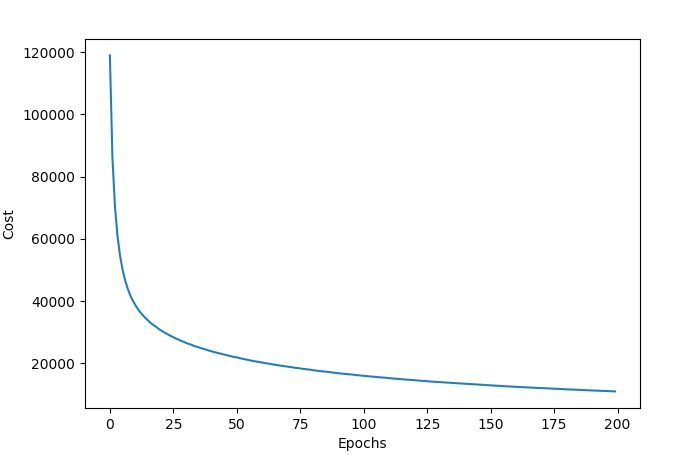
* Original Accuracies
  + Train: 99.28%
  + Valid: 97.98%
  + Test: 97.54%
* Experiments
  + Part One – Learning Rate Hyperparameter
    - By increasing the learning rate from 0.0001 to 0.00015, an increase in convergence could be seen as the train accuracy was 97.57% and the valid accuracy was 97.38%. The test accuracy with the increased learning rate of 0.00015 was 96.63%. In conclusion, you can indeed improve the accuracy of the network by modifying the learning rate hyperparameter.

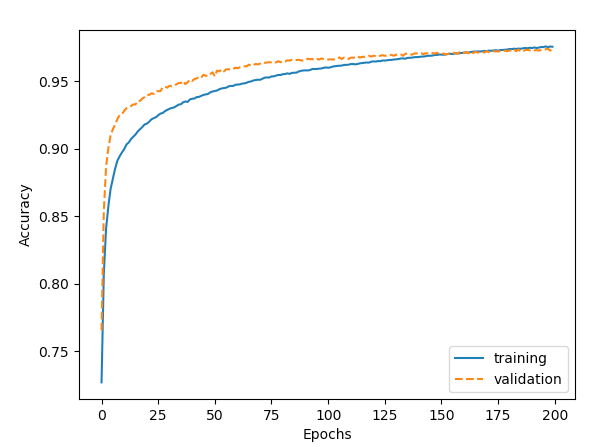


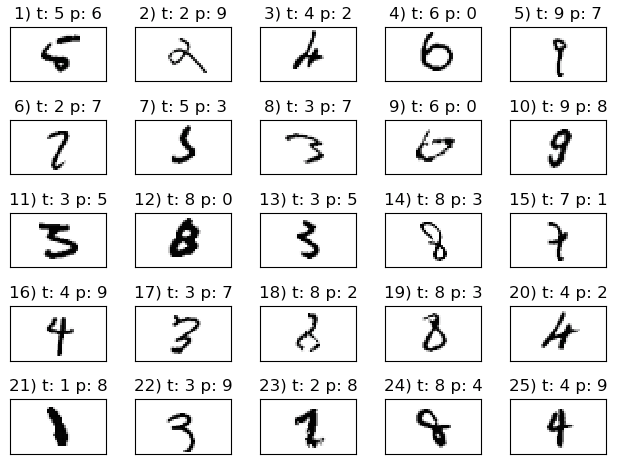




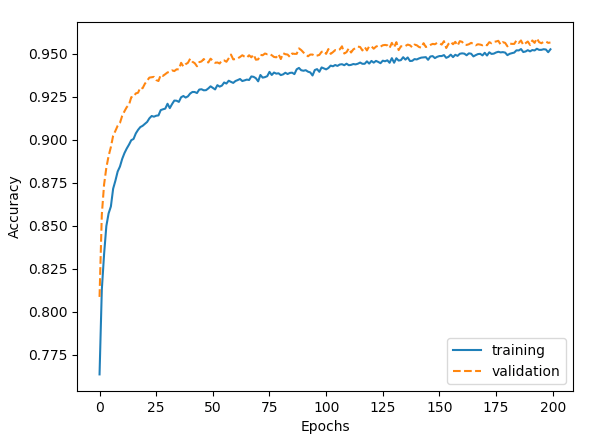
* + Part Two – Minibatch Size
    - By decreasing the minibatch size from 100 to 89 an improvement in accuracy to the network resulted. The train accuracy improved in terms of convergence to 97.55% after the modification to the minibatch size compared to 97.57%. The valid accuracy of the network remained the same at 97.38%, showing an increase in convergence. The test accuracy of the network also improved to 96.63% after the minibatch modification compared to 96.61% prior to the modification. In conclusion, you can indeed improve the accuracy of the network by modifying the minibatch size.

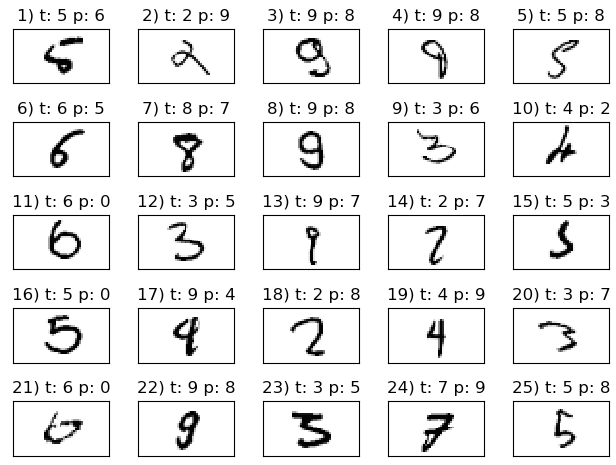




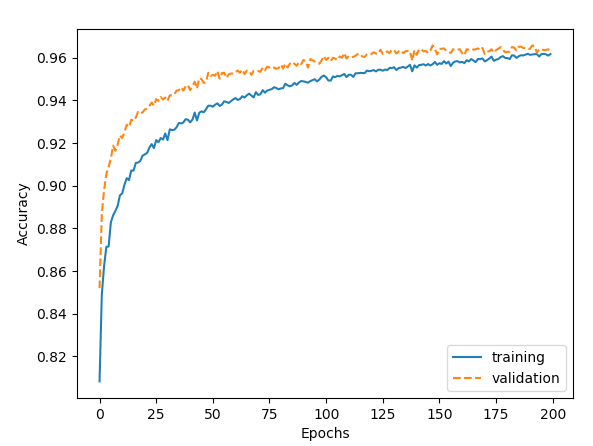


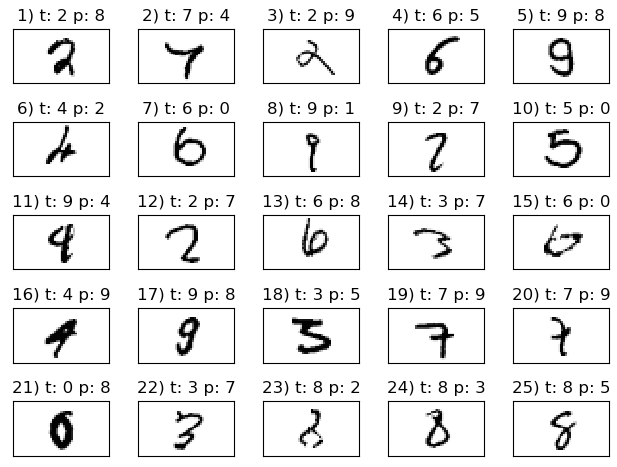
* + Part Three – Tanh Function vs. Sigmoid Function
    - By utilizing the tanh function the accuracy of the network did not improve, it in fact decreased. With the use of tanh the train accuracy was reduced to 95.27% compared to 97.55% with sigmoid function. The valid accuracy dropped to 95.66% with the use of the tanh function compared to 97.38% with the use of sigmoid function. The test accuracy also dropped with the use of the tanh function to 94.53% compared to 96.63% with the use of sigmoid function. In conclusion, the use of tanh does not improve the accuracy of the network.





* + Part Four – Modifying Number of Neurons in First Layer
    - By increasing the number of neurons in the first layer, you can indeed increase the accuracy of the network. I found the optimal number of neurons to be 200 for the network. 200 neurons in the first layer yielded a train accuracy of 96.17%, a valid accuracy of 96.34%, and a test accuracy of 95.32%. These are all improvements over the original network, which utilized 100 neurons in the first layer yielding a train accuracy of 95.26%, a valid accuracy of 95.66%, and a test accuracy of 94.53%. I also attempted to use 150 neurons in the first layer which yield a train accuracy of 95.87%, a valid accuracy of 96.02%, and a test accuracy of 95.14%, these figures further confirm the trend that increasing the number of neurons in the first layer can increase the accuracy of the network. I also found that decreasing the number of neurons in the first layer also decreases the accuracy of the network. Attempting to run with 30 neurons in the first layer resulted in a train accuracy of 92.27%, a valid accuracy of 92.01%, and a test accuracy of 93.18%, all lower than the accuracies of the original setting of 100 neurons. In conclusion, increasing the number of neurons in the first layer yields an increase in the accuracy of the network, as the optimal number of neurons I found was 200.





* + Part Five – Topology
    - Did not complete.
* The Modified Program

# Kyle Calabro

# Dr. Kumar

# Artificial Intelligence - Project 5

# 29 April 2018

import os

import struct

import numpy as np

# To import the mnist database

def load\_mnist(path, kind='train'):

"""Load MNIST data from `path`"""

labels\_path = os.path.join(path,

'%s-labels-idx1-ubyte' % kind)

images\_path = os.path.join(path,

'%s-images-idx3-ubyte' % kind)

with open(labels\_path, 'rb') as lbpath:

magic, n = struct.unpack('>II',

lbpath.read(8))

labels = np.fromfile(lbpath,

dtype=np.uint8)

with open(images\_path, 'rb') as imgpath:

magic, num, rows, cols = struct.unpack(">IIII",

imgpath.read(16))

images = np.fromfile(imgpath,

dtype=np.uint8).reshape(len(labels), 784)

images = ((images / 255.) - .5) \* 2

return images, labels

# Unzipping the mnist database

import sys

import gzip

import shutil

if (sys.version\_info > (3, 0)):

writemode = 'wb'

else:

writemode = 'w'

zipped\_mnist = [f for f in os.listdir('./') if f.endswith('ubyte.gz')]

for z in zipped\_mnist:

with gzip.GzipFile(z, mode='rb') as decompressed, open(z[:-3], writemode) as outfile:

outfile.write(decompressed.read())

# Load training records

X\_train, y\_train = load\_mnist('', kind='train')

print('Rows: %d, columns: %d' % (X\_train.shape[0], X\_train.shape[1]))

# Load testing records

X\_test, y\_test = load\_mnist('', kind='t10k')

print('Rows: %d, columns: %d' % (X\_test.shape[0], X\_test.shape[1]))

# Visualize the first digit of each class:

import matplotlib.pyplot as plt

fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True,)

ax = ax.flatten()

for i in range(10):

img = X\_train[y\_train == i][0].reshape(28, 28)

ax[i].imshow(img, cmap='Greys')

ax[0].set\_xticks([])

ax[0].set\_yticks([])

plt.tight\_layout()

plt.show()

# Visualize 25 different versions of "7":

fig, ax = plt.subplots(nrows=5, ncols=5, sharex=True, sharey=True,)

ax = ax.flatten()

for i in range(25):

img = X\_train[y\_train == 7][i].reshape(28, 28)

ax[i].imshow(img, cmap='Greys')

ax[0].set\_xticks([])

ax[0].set\_yticks([])

plt.tight\_layout()

plt.show()

# To save and restore - optional

np.savez\_compressed('mnist\_scaled.npz',

X\_train=X\_train,

y\_train=y\_train,

X\_test=X\_test,

y\_test=y\_test)

mnist = np.load('mnist\_scaled.npz')

mnist.files

X\_train, y\_train, X\_test, y\_test = [mnist[f] for f in ['X\_train', 'y\_train',

'X\_test', 'y\_test']]

del mnist

X\_train.shape

# -------------------------------------------------------------------------------#

# Implementing a multi-layer perceptron

class NeuralNetMLP(object):

""" Feedforward neural network / Multi-layer perceptron classifier.

Parameters

------------

n\_hidden : int (default: 30)

Number of hidden units.

l2 : float (default: 0.)

Lambda value for L2-regularization.

No regularization if l2=0. (default)

epochs : int (default: 100)

Number of passes over the training set.

eta : float (default: 0.001)

Learning rate.

shuffle : bool (default: True)

Shuffles training data every epoch if True to prevent circles.

minibatche\_size : int (default: 1)

Number of training samples per minibatch.

seed : int (default: None)

Random seed for initalizing weights and shuffling.

Attributes

-----------

eval\_ : dict

Dictionary collecting the cost, training accuracy,

and validation accuracy for each epoch during training.

"""

def \_\_init\_\_(self, n\_hidden=30,

l2=0., epochs=100, eta=0.001,

shuffle=True, minibatch\_size=1, seed=None):

self.random = np.random.RandomState(seed)

self.n\_hidden = n\_hidden

self.l2 = l2

self.epochs = epochs

self.eta = eta

self.shuffle = shuffle

self.minibatch\_size = minibatch\_size

def \_onehot(self, y, n\_classes):

"""Encode labels into one-hot representation

Parameters

------------

y : array, shape = [n\_samples]

Target values.

Returns

-----------

onehot : array, shape = (n\_samples, n\_labels)

"""

onehot = np.zeros((n\_classes, y.shape[0]))

for idx, val in enumerate(y.astype(int)):

onehot[val, idx] = 1.

return onehot.T

def \_sigmoid(self, z):

"""Compute logistic function (sigmoid)"""

return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

def \_tanh(self, z):

e\_p = np.exp(z)

e\_m = np.exp(-z)

return (e\_p - e\_m) / (e\_p + e\_m)

def \_forward(self, X):

"""Compute forward propagation step"""

# step 1: net input of hidden layer

# [n\_samples, n\_features] dot [n\_features, n\_hidden]

# -> [n\_samples, n\_hidden]

z\_h = np.dot(X, self.w\_h) + self.b\_h

# step 2: activation of hidden layer

# a\_h = self.\_sigmoid(z\_h)

a\_h = self.\_tanh(z\_h)

# step 3: net input of output layer

# [n\_samples, n\_hidden] dot [n\_hidden, n\_classlabels]

# -> [n\_samples, n\_classlabels]

z\_out = np.dot(a\_h, self.w\_out) + self.b\_out

# step 4: activation output layer

#a\_out = self.\_sigmoid(z\_out)

a\_out = self.\_tanh(z\_out)

return z\_h, a\_h, z\_out, a\_out

def \_compute\_cost(self, y\_enc, output):

"""Compute cost function.

Parameters

----------

y\_enc : array, shape = (n\_samples, n\_labels)

one-hot encoded class labels.

output : array, shape = [n\_samples, n\_output\_units]

Activation of the output layer (forward propagation)

Returns

---------

cost : float

Regularized cost

"""

L2\_term = (self.l2 \*

(np.sum(self.w\_h \*\* 2.) +

np.sum(self.w\_out \*\* 2.)))

term1 = -y\_enc \* (np.log(output))

term2 = (1. - y\_enc) \* np.log(1. - output)

cost = np.sum(term1 - term2) + L2\_term

return cost

def \_tanh\_derivative(self, z):

#e\_p = np.exp(z)

#e\_m = np.exp(-z)

#return (2 / (e\_p + e\_m))\*\*2

return 1.0 - np.tanh(z)\*\*2

def predict(self, X):

"""Predict class labels

Parameters

-----------

X : array, shape = [n\_samples, n\_features]

Input layer with original features.

Returns:

----------

y\_pred : array, shape = [n\_samples]

Predicted class labels.

"""

z\_h, a\_h, z\_out, a\_out = self.\_forward(X)

y\_pred = np.argmax(z\_out, axis=1)

return y\_pred

def fit(self, X\_train, y\_train, X\_valid, y\_valid):

""" Learn weights from training data.

Parameters

-----------

X\_train : array, shape = [n\_samples, n\_features]

Input layer with original features.

y\_train : array, shape = [n\_samples]

Target class labels.

X\_valid : array, shape = [n\_samples, n\_features]

Sample features for validation during training

y\_valid : array, shape = [n\_samples]

Sample labels for validation during training

Returns:

----------

self

"""

n\_output = np.unique(y\_train).shape[0] # number of class labels

n\_features = X\_train.shape[1]

########################

# Weight initialization

########################

# weights for input -> hidden

self.b\_h = np.zeros(self.n\_hidden)

self.w\_h = self.random.normal(loc=0.0, scale=0.1,

size=(n\_features, self.n\_hidden))

# weights for hidden -> output

self.b\_out = np.zeros(n\_output)

self.w\_out = self.random.normal(loc=0.0, scale=0.1,

size=(self.n\_hidden, n\_output))

epoch\_strlen = len(str(self.epochs)) # for progress formatting

self.eval\_ = {'cost': [], 'train\_acc': [], 'valid\_acc': []}

y\_train\_enc = self.\_onehot(y\_train, n\_output)

# iterate over training epochs

for i in range(self.epochs):

# iterate over minibatches

indices = np.arange(X\_train.shape[0])

if self.shuffle:

self.random.shuffle(indices)

for start\_idx in range(0, indices.shape[0] - self.minibatch\_size +

1, self.minibatch\_size):

batch\_idx = indices[start\_idx:start\_idx + self.minibatch\_size]

# forward propagation

z\_h, a\_h, z\_out, a\_out = self.\_forward(X\_train[batch\_idx])

##################

# Backpropagation

##################

# [n\_samples, n\_classlabels]

sigma\_out = a\_out - y\_train\_enc[batch\_idx]

# [n\_samples, n\_hidden]

sigmoid\_derivative\_h = a\_h \* (1. - a\_h)

# [n\_samples, n\_classlabels] dot [n\_classlabels, n\_hidden]

# -> [n\_samples, n\_hidden]

#sigma\_h = (np.dot(sigma\_out, self.w\_out.T) \*

# sigmoid\_derivative\_h)

sigma\_h = (np.dot(sigma\_out, self.w\_out.T) \*

self.\_tanh\_derivative(a\_h))

# [n\_features, n\_samples] dot [n\_samples, n\_hidden]

# -> [n\_features, n\_hidden]

grad\_w\_h = np.dot(X\_train[batch\_idx].T, sigma\_h)

grad\_b\_h = np.sum(sigma\_h, axis=0)

# [n\_hidden, n\_samples] dot [n\_samples, n\_classlabels]

# -> [n\_hidden, n\_classlabels]

grad\_w\_out = np.dot(a\_h.T, sigma\_out)

grad\_b\_out = np.sum(sigma\_out, axis=0)

# Regularization and weight updates

delta\_w\_h = (grad\_w\_h + self.l2\*self.w\_h)

delta\_b\_h = grad\_b\_h # bias is not regularized

self.w\_h -= self.eta \* delta\_w\_h

self.b\_h -= self.eta \* delta\_b\_h

delta\_w\_out = (grad\_w\_out + self.l2\*self.w\_out)

delta\_b\_out = grad\_b\_out # bias is not regularized

self.w\_out -= self.eta \* delta\_w\_out

self.b\_out -= self.eta \* delta\_b\_out

#############

# Evaluation

#############

# Evaluation after each epoch during training

z\_h, a\_h, z\_out, a\_out = self.\_forward(X\_train)

cost = self.\_compute\_cost(y\_enc=y\_train\_enc,

output=a\_out)

y\_train\_pred = self.predict(X\_train)

y\_valid\_pred = self.predict(X\_valid)

train\_acc = ((np.sum(y\_train == y\_train\_pred)).astype(np.float) /

X\_train.shape[0])

valid\_acc = ((np.sum(y\_valid == y\_valid\_pred)).astype(np.float) /

X\_valid.shape[0])

sys.stderr.write('\r%0\*d/%d | Cost: %.2f '

'| Train/Valid Acc.: %.2f%%/%.2f%% ' %

(epoch\_strlen, i+1, self.epochs, cost,

train\_acc\*100, valid\_acc\*100))

sys.stderr.flush()

self.eval\_['cost'].append(cost)

self.eval\_['train\_acc'].append(train\_acc)

self.eval\_['valid\_acc'].append(valid\_acc)

return self

n\_epochs = 200

nn = NeuralNetMLP(n\_hidden=150,

l2=0.01,

epochs=n\_epochs,

eta=0.00015,

minibatch\_size=89,

shuffle=True,

seed=1)

nn.fit(X\_train=X\_train[:55000],

y\_train=y\_train[:55000],

X\_valid=X\_train[55000:],

y\_valid=y\_train[55000:])

import matplotlib.pyplot as plt

plt.plot(range(nn.epochs), nn.eval\_['cost'])

plt.ylabel('Cost')

plt.xlabel('Epochs')

plt.show()

plt.plot(range(nn.epochs), nn.eval\_['train\_acc'],

label='training')

plt.plot(range(nn.epochs), nn.eval\_['valid\_acc'],

label='validation', linestyle='--')

plt.ylabel('Accuracy')

plt.xlabel('Epochs')

plt.legend()

plt.show()

y\_test\_pred = nn.predict(X\_test)

acc = (np.sum(y\_test == y\_test\_pred)

.astype(np.float) / X\_test.shape[0])

print('Test accuracy: %.2f%%' % (acc \* 100))

miscl\_img = X\_test[y\_test != y\_test\_pred][:25]

correct\_lab = y\_test[y\_test != y\_test\_pred][:25]

miscl\_lab = y\_test\_pred[y\_test != y\_test\_pred][:25]

fig, ax = plt.subplots(nrows=5, ncols=5, sharex=True, sharey=True,)

ax = ax.flatten()

for i in range(25):

img = miscl\_img[i].reshape(28, 28)

ax[i].imshow(img, cmap='Greys', interpolation='nearest')

ax[i].set\_title('%d) t: %d p: %d' % (i+1, correct\_lab[i], miscl\_lab[i]))

ax[0].set\_xticks([])

ax[0].set\_yticks([])

plt.tight\_layout()

plt.show()