Group 35

Name: Shaokat Hossain

Id: 1144107

Name: Md Abu Noman Majumdar

Id: 1144101

Name: Abdullah Al Amin

Id: 1152610

Setup for this exercise sheet. Download data and define Tensorflow version. Execute code only if you setup your enviroment correctly or if you are inside a colab enviroment.

In [ ]:
```
# ! git clone https://gitlab+deploy-token-26:XBza882znMmexaQSpjad@git.informatik.

# %tensorflow_version 2.x
```

Exercise 1 (Learning in neural networks)

a) Explain the following terms related to neural networks as short and precise as possible.

- Loss function
- Stochastic gradient descent
- Mini-batch
- Regularization
- Dropout
- Batch normalization
- Learning with momentum
- Data augmentation
- Unsupervised pre-training / supervised fine-tuning
- Deep learning

Loss Function: A loss function is used to optimize the parameter values in a neural network model. Loss functions map a set of parameter values for the network onto a scalar value that indicates how well those parameters accomplish the task the network is intended to do.

Stochastic gradient descent: Stochastic gradient descent is a method to find the optimal parameter configuration for a machine learning algorithm. It iteratively makes small adjustments to a machine learning network configuration to decrease the error of the network. Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or subdifferentiable). It can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient

(calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in big data applications this reduces the computational burden, achieving faster iterations in trade for a slightly lower convergence rate.

Mini-batch: It is actually a compromise between full-batch iteration and generalized SGD. Often regarded as a small subset of the training data. The size of the mini batch is chosen according to hardware resources and algorithmic requirements. A mini-batch is typically between 10 and 1000 examples chosen at random.

Regularization: This is a form of regression, that constrains/ regularizes or shrinks the coefficient estimates towards zero. In other words, this technique discourages learning a more complex or flexible model, so as to avoid the risk of overfitting.. Regularization is a technique which makes slight modifications to the learning algorithm and generalizes better. This in turn improves the model's performance on the unseen data as well.

Dropout: Dropout is a regularization technique to reduce the risk of overfitting, mostly applied in fully connected layers of neural networks. This is one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning. At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections. So each iteration has a different set of nodes and this results in a different set of outputs. It can also be thought of as an ensemble technique in machine learning. Ensemble models usually perform better than a single model as they capture more randomness. Similarly, dropout also performs better than a normal neural network model This probability of choosing how many nodes should be dropped is the hyperparameter of the dropout function. Dropout can be applied to both the hidden layers as well as the input layers. Dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

Batch-Normalization: We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers that are changing all the time, and get 10 ties or more improvement in the training speed. Batch normalization reduces the amount by what the hidden unit values shift around.

Learning with momentum: Momentum methods in the context of machine learning refer to a group of tricks and techniques designed to speed up convergence of first order optimization methods like gradient descent (and its many variants).They essentially work by adding what's called the momentum term to the update formula for gradient descent, thereby ameliorating its natural "zigzagging behavior," especially in long narrow valleys of the cost function. Neural network momentum is a simple technique that often improves both training speed and accuracy. Training a neural network is the process of finding values for the weights and biases so that for a given set of input values, the computed output values closely match the known, correct, target values. Learning with momentum is a technique applied in gradient descent learning to improve convergence. For small learning rates, gradient descent based learning is too large, the weight update may overshoot, leading to an oscillating loss function.

Data augmentation: We do augmentation before we feed the data to the model. We have two options. One option is to perform all the necessary transformations beforehand, essentially increasing the size of our dataset. The other option is to perform these transformations on a mini-

batch, just before feeding it to our machine learning model. The first option is known as offline augmentation. This method is preferred for relatively smaller datasets, as we would end up increasing the size of the dataset by a factor equal to the number of transformations we perform. The second option is known as online augmentation, or augmentation on the fly. This method is preferred for larger datasets, as we can't afford the explosive increase in size. Instead, we would perform transformations on the mini-batches that we would feed to our model. Some machine learning frameworks have support for online augmentation, which can be accelerated on the GPU.

Unsupervised pre training/supervised fine-tuning: Training deep feed-forward neural networks can be difficult because of local optima in the objective function and because complex models are prone to overfitting. Unsupervised pre-training initializes as discriminative neural net from one which was trained using an unsupervised criterion, such as a deep belief network or a deep autoencoder. This method can sometimes help with both the optimization and the overfitting issues.

Deep learning:Deep learning is an artificial intelligence function that imitates the workings of the human brain in processing data and creating patterns for use in decision making. Deep learning is a subset of machine learning in artificial intelligence (AI) that has networks capable of learning unsupervised from data that is unstructured or unlabeled. Also known as deep neural learning or deep neural network.

b) Name the most important output activation functions f(z), i.e., activation function of the output neuron(s), together with a corresponding suitable loss function L (in both cases, give the mathematical equation). Indicate whether such a perceptron is used for a classification or a regression task.

Most import activation Function:

Binary Step Function:

Mathematical Equation: $f(x) = 0$ if $x >= 0$

Use case: It is used while creating a binary classifier.

Linear Function:

Mathematical Equation: $f(x) = ax$

Use case: It is used for Regression related tasks.

Sigmoid Function:

Mathematical Equation: $f(x) = \frac{1}{1+e^{-x}}$

Loss function: Binary Cross Entropy

Mathematical Equation:

Use case: classify the values to particular classes.

Tanh:

Mathematical Equation: $tanh(x) = \frac{2}{1+e^{-2x}-1}$

Use case: Classification.

ReLU:

Mathematical Equation: $f(x) = max(0, x)$

Loss function:Mean squared error (MSE)

Use case: Regression, but Non-Negative.

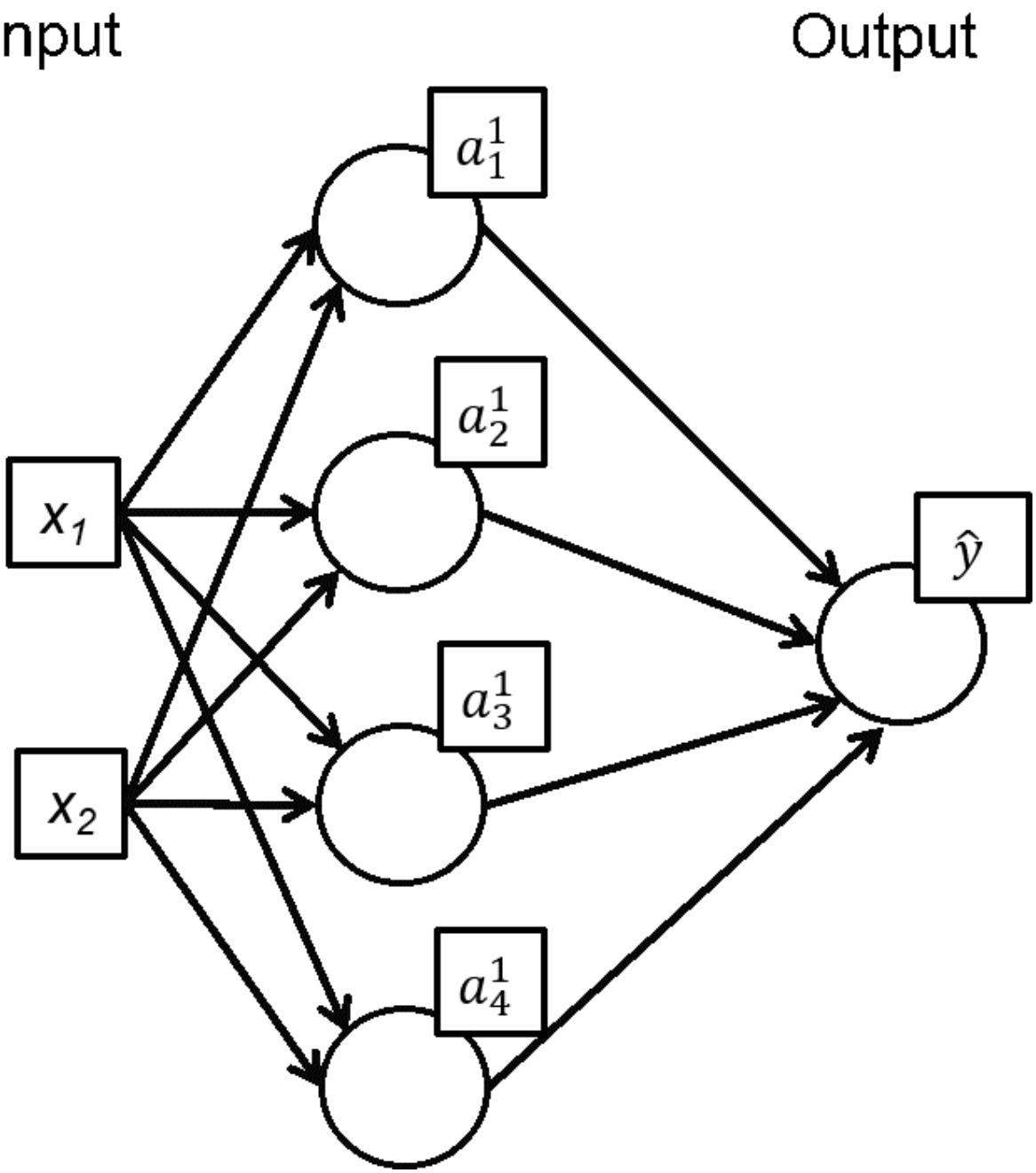
Softmax:

Loss function:Cross Entropy

Use case: Classification.

Exercise 2 (Multi-layer perceptron: Backprogagation, regression problem)

a) Consider the multi-layer perceptron in the following figure:



The activation function at all hidden nodes is ReLU and at the output node linear.

Perform one iteration of plain backpropagation (without momentum, regularization etc.), based on a mini-batch composed of two input samples $x^{(\mu)}$ with corresponding target values $y^{(\mu)}$, learning rate η and SSE loss:

$x^{(1)} = (-1, 1)^T$ with target $y^{(1)} = 1$ and $x^{(2)} = (2, -1)^T$ with target $y^{(2)} = -1$

The initial weights and biases are given as (t is the iteration index):

$$W^1(t=0) = \begin{bmatrix} 1 & 2 \\ 0 & -1 \\ -1 & -3 \\ -2 & 2 \end{bmatrix}; W^2(t=0) = \begin{bmatrix} 1 & 0 & -1 & 2 \end{bmatrix}$$

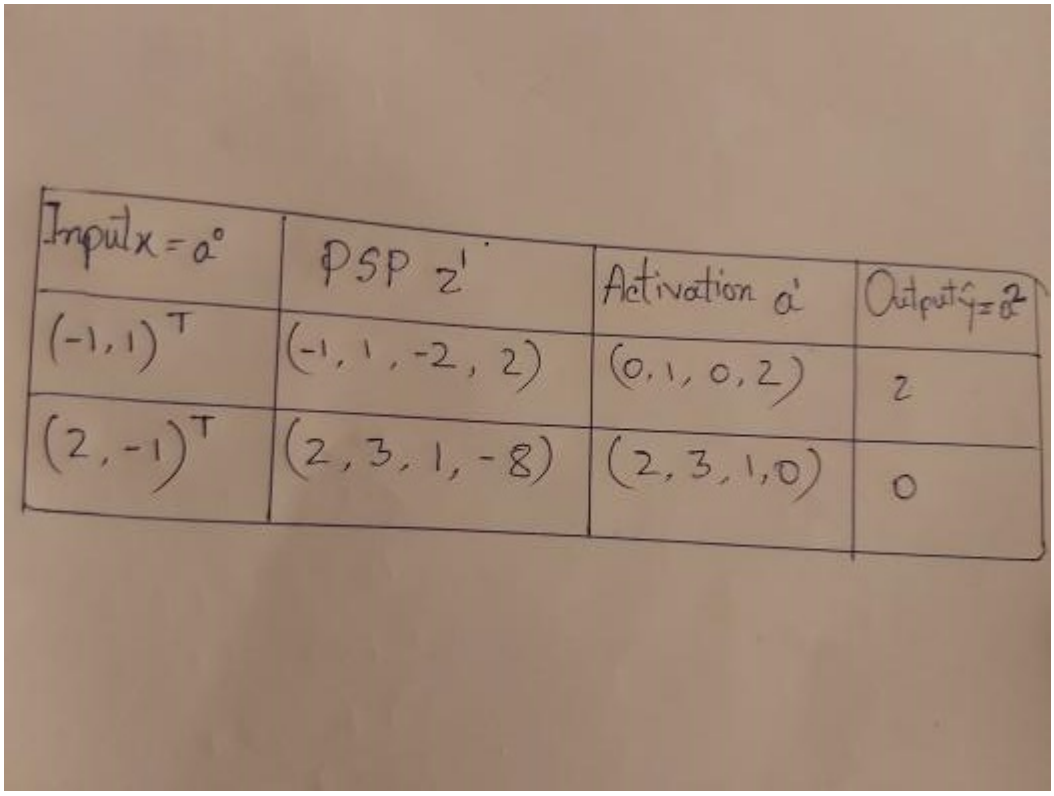
$$b^1(t=0) = \begin{bmatrix} -2 \\ 2 \\ 0 \\ -2 \end{bmatrix}; b^2(t=0) = -2$$

For the forward path, calculate the postsynaptic potential (PSP), the activations and outputs and insert them into the following table:

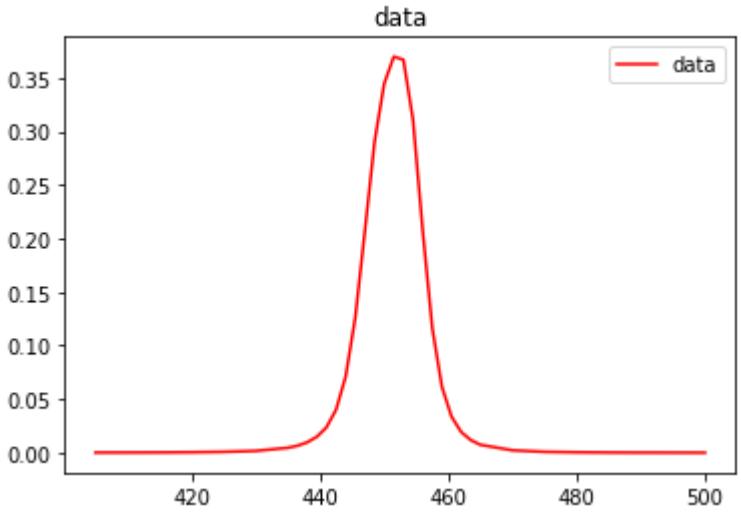
Input $x = (x_1, x_2)^T = a^0$	**PSP** z^1	**Activation** a^1	**Ouput** $\hat{y} = a^2$
$(-1, 1)^T$			
$(2, -1)^T$			

For the backward path, calculate the updated weights and biases for the hidden and output layer and insert them into the following table:

Weights $W^1(t=1)$	**Bias** $b^1(t=1)$	**Weights** $W^2(t=1)$	**Bias** $b^2(t=1)$



b) The goal of this exercise is to train a multi-layer perceptron to solve a high difficulty level nonlinear regression problem. The data has been generated using an exponential function with the following shape:



This graph corresponds to the values of a dataset that can be downloaded from the Statistical Reference Dataset of the Information Technology Laboratory of the United States on this link: http://www.itl.nist.gov/div898/strd/nls/data/eckerle4.shtml (http://www.itl.nist.gov/div898/strd/nls/data/eckerle4.shtml)

This dataset is provided in the file Eckerle4.csv. Note that this dataset is divided into a training and test corpus comprising 60% and 40% of the data samples, respectively. Moreover, the input and output values are normalized to the interval [0, 1]. Basic code to load the dataset and divide it into a training and test corpus, normalizing the data and to apply a multi-layer perceptron is provided in the Jupyter notebook.

Choose a suitable network topology (number of hidden layers and hidden neurons, potentially include dropout, activation function of hidden layers) and use it for the multi-layer perceptron defined in the Jupyter notebook. Set further parameters (learning rate, loss function, optimizer, number of epochs, batch size; see the lines marked with *# FIX!!!* in the Jupyter notebook). Try to avoid underfitting and overfitting. Vary the network and parameter configuration in order to achieve a network performance as optimal as possible. For each network configuration, due to the random components in the experiment, perform (at least) 4 different training and evaluation runs and report the mean and standard deviation of the training and evaluation results. Report on your results and conclusions.

(Source of exercise: http://gonzalopla.com/deep-learning-nonlinear-regression (http://gonzalopla.com/deep-learning-nonlinear-regression))

In [ ]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from os.path import join
from tensorflow.keras.layers import Dense, Dropout, Activation
from tensorflow.keras import Model, Input, Sequential
from tensorflow.keras.optimizers import SGD, Adam, Adadelta, Adagrad, Nadam, RMSp
from tensorflow.keras.utils import normalize
import pandas
from sklearn import preprocessing
from sklearn import model_selection
import sys

###--------
# load data
###--------

# Imports csv into pandas DataFrame object.
path_to_task = "nndl/Lab4"
Eckerle4_df = pandas.read_csv(join(path_to_task,"Eckerle4.csv"), header=0)

# Converts dataframes into numpy objects.
Eckerle4_dataset = Eckerle4_df.values.astype("float32")
# Slicing all rows, second column...
X = Eckerle4_dataset[:,1]
# Slicing all rows, first column...
y = Eckerle4_dataset[:,0]

# plot data
plt.plot(X,y, color='red')
plt.legend(labels=["data"], loc="upper right")
plt.title("data")
plt.show()

###-----------
# process data
###-----------

# Data Scaling from 0 to 1, X and y originally have very different scales.
X_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
y_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
X_scaled = ( X_scaler.fit_transform(X.reshape(-1,1)))
y_scaled = (y_scaler.fit_transform(y.reshape(-1,1)).reshape(-1) )

# Preparing test and train data: 60% training, 40% testing.
X_train, X_test, y_train, y_test = model_selection.train_test_split( X_scaled, y_


###-----------
# define model
###-----------

num_inputs = X_train.shape[1] # should be 1 in case of Eckerle4
num_hidden = ... # for each hidden layer: number of hidden units in form of a pyt
num_outputs = 1 # predict single number in case of Eckerle4
```

```
activation = '...' # activation of hidden layers   # FIX!!!
dropout = ... # 0 if no dropout, else fraction of dropout units (e.g. 0.2)   # FI

# Sequential network structure.
model = Sequential()

if len(num_hidden) == 0:
  print("Error: Must at least have one hidden layer!")
  sys.exit()

# add first hidden layer connecting to input layer
model.add(Dense(num_hidden[0], input_dim=num_inputs, activation=activation))

if dropout:
  # dropout of fraction dropout of the neurons and activation layer.
  model.add(Dropout(dropout))
  # model.add(Activation("linear"))

# potentially further hidden layers
for i in range(1, len(num_hidden)):
  # add hidden layer with len[i] neurons
  model.add(Dense(num_hidden[i], activation=activation))
#  model.add(Activation("linear"))

  if dropout:
  # dropout of fraction dropout of the neurons and activation layer.
    model.add(Dropout(dropout))
  #  model.add(Activation("linear"))

# output layer
model.add(Dense(1))

# show how the model looks
model.summary()

# compile model
opt = ... # FIX!!!
model.compile(loss='...', optimizer=opt, metrics=["..."])# FIX!!!

# Training model with train data. Fixed random seed:
np.random.seed(3)
num_epochs = ...   # FIX !!!
batch_size = ... # FIX !!!
history = model.fit(X_train, y_train, epochs=num_epochs, batch_size=batch_size, v

###-----------
# plot results
###-----------

print("final (mse) training error: %f" % history.history['loss'][num_epochs-1])

plt.plot(history.history['loss'], color='red', label = 'training loss')
plt.legend(labels=["loss"], loc="upper right")
plt.title("training (mse) error")
plt.show()

# Plot in blue color the predicted data and in green color the
```

```
# actual data to verify visually the accuracy of the model.
predicted = model.predict(X_test)
plt.plot(y_scaler.inverse_transform(predicted.reshape(-1,1)), color="blue")
plt.plot(y_scaler.inverse_transform(y_test.reshape(-1,1)), color="green")
plt.legend(labels=["predicted", "target"], loc="upper right")
plt.title("evaluation on test corpus")
plt.show()
print("test error: %f" % model.evaluate(X_test, y_test)[0])
```

Experiment Info and results

num_epochs = 256
batch_size = 5

opt = 'adam' criterion = 'loss' mean = 0.028025922319342443 std = 0.02521123848194273

criterion = 'accuracy' mean = 0.05 std = 0.05

opt = 'SGD'

criterion = 'loss' mean = 0.07073916167209973 std = 0.005167926836764933

criterion = 'accuracy' mean = 0.05 std = 0.05

opt = 'RMSProps'

criterion = 'loss' mean = 0.034314523537902875 std = 0.02346860155992621

criterion = 'accuracy' mean = 0.05 std = 0.05

Exercise 3 (Parameters of a multi-layer perceptron – digit recognition)

In the following exercises, we use Tensorflow and Keras to configure, train and apply a multi-layer perceptron to the problem of recognizing handwritten digits (the famous “MNIST” problem). The MNIST data are loaded using a Tensorflow Keras built-in function.

Perform experiments on this pattern recognition problem trying to investigate the influence of a number of parameters on the classification performance. This may refer to

- the learning rate and potentially learning schedule,
- the number of hidden neurons (in a network with a single hidden layer),
- the number of hidden layers as well as applying dropout and / or batch normalization,
- the solver (including momentum),
- the activation function at hidden layers,
- regularization.

The script in the Jupyter notebook can serve as a basis or starting point.

Report your findings and conclusions.

Note: These experiments may require a lot of computation time!

Further investigations and experiments as well as code extensions and modifications are welcome!

In [ ]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from os.path import join
from tensorflow.keras.layers import Dense, Dropout, Activation, BatchNormalizatic
from tensorflow.keras import Model, Input, Sequential
from tensorflow.keras.optimizers import SGD, Adam, Adadelta, Adagrad, Nadam, RMSp
from tensorflow.keras.utils import normalize
import tensorflow.keras.datasets as tfds
import tensorflow.keras.initializers as tfi
import tensorflow.keras.regularizers as tfr

###--------
# load data
###--------

(training_input, training_target), (test_input, test_target)  = tfds.mnist.load_d

# Reserve 10,000 samples for validation
validation_input = training_input[-10000:]
validation_target = training_target[-10000:]
training_input = training_input[:-10000]
training_target = training_target[:-10000]

print("training input shape: %s, training target shape: %s"  % (training_input.sh
print("validation input shape: %s, validation target shape: %s"  % (validation_in
print("test input shape: %s, test target shape: %s"  % (test_input.shape, test_ta
# range of input values: 0 ... 255
print("\n")

# plot some sample images
num_examples = 2
for s in range(num_examples):
  print("Example image, true label: %d" % training_target[s])
  plt.imshow(training_input[s], vmin=0, vmax=255, cmap=plt.cm.gray)
  plt.show()

###-----------
# process data
###-----------

# Note: shuffling is performed in fit method

# scaling inputs from range 0 ... 255 to range [0,1] if desired
scale_inputs = True # scale inputs to range [0,1]
if scale_inputs:
  training_input = training_input / 255
  validation_input = validation_input / 255
  test_input = test_input / 255

print("min. training data: %f" % np.min(training_input))
print("max. training data: %f" % np.max(training_input))
print("min. validation data: %f" % np.min(validation_input))
print("max. validation data: %f" % np.max(validation_input))
print("min. test data: %f" % np.min(test_input))
print("max. test data: %f" % np.max(test_input))
```

```
# histograms of input values
nBins = 100
fig, axes = plt.subplots(1, 3, figsize=(15,10))
axes[0].hist(training_input.flatten(), nBins)
axes[0].set_xlabel("training")
axes[0].set_ylabel("counts")
axes[0].set_ylim((0,1e6))

axes[1].hist(validation_input.flatten(), nBins)
axes[1].set_xlabel("validation")
axes[1].set_ylabel("counts")
axes[1].set_ylim((0,1e6))
axes[1].set_title('historgrams of input values')

axes[2].hist(test_input.flatten(), nBins)
axes[2].set_xlabel("test")
axes[2].set_ylabel("counts")
axes[2].set_ylim((0,1e6))

plt.show()

# flatten inputs to vectors
training_input = training_input.reshape(training_input.shape[0], training_input.s
validation_input = validation_input.reshape(validation_input.shape[0], validatior
test_input = test_input.reshape(test_input.shape[0], test_input.shape[1] * test_i
print(training_input.shape)
print(validation_input.shape)
print(test_input.shape)

num_classes = len(np.unique(training_target)) # FIX!!!

###-----------
# define model
###-----------

histories = {}
opt_learning_rate = {}
final_training_loss = {}
final_training_accuracy = {}
final_validation_loss = {}
final_validation_accuracy = {}
final_test_loss = {}
final_test_accuracy = {}

configurations = [
        # FIX!!!
        # ...
#         {'learningRates': [0.11,0.13], # numpy array, e.g. [0.1, 0.2]
#          'hiddenLayerSizes': [60,30], # as before
#          'solver': 'SGD',
#          'activation':'relu'}, # activation of hidden layers
         {'learningRates': [0.2,0.25], # numpy array, e.g. [0.1, 0.2]
             'hiddenLayerSizes': [220], # as before
             'solver': 'Adam',
             'activation':'relu'}, # activation of hidden layers
        {'learningRates': [0.2,0.25], # numpy array, e.g. [0.1, 0.2]
```

```
                'hiddenLayerSizes': [220], # as before
                'solver': 'Nadam',
                'activation':'relu'}, # activation of hidden layers
#         {'learningRates': [0.11,0.13], # numpy array, e.g. [0.1, 0.2]
#                 'hiddenLayerSizes': [70,60], # as before
#                 'solver': 'Adagrad',
#                 'activation':'relu'}, # activation of hidden layers
#         {'learningRates': [0.11,0.13], # numpy array, e.g. [0.1, 0.2]
#                 'hiddenLayerSizes': [70,70], # as before
#                 'solver': 'Adadelta',
#                 'activation':'relu'}, # activation of hidden layers
         {'learningRates': [0.11,0.13], # numpy array, e.g. [0.1, 0.2]
                'hiddenLayerSizes': [220], # as before
                'solver': 'RMSProp',
                'activation':'relu'}, # activation of hidden layers


]

learningRateSchedule = False # FIX!!! True: apply (exponential) learning rate sch
dropout = 0.21 # FIX!!! 0 if no dropout, else fraction of dropout units (e.g. 0.2
batch_normalization = False # FIX!!!
regularization_weight = 0.01 # FIX!!! 0 for no regularization or e.g. 0.01 to app
regularizer = tfr.l1(l=regularization_weight) # or l2 or l1_l2; used for both wei
momentum = 0.9 # FIX!!! 0 or e.g. 0.9, 0.99; ONLY FOR STOCHASTIC GRADIENT DESCENT
nesterov = True # FIX!!! ONLY FOR STOCHASTIC GRADIENT DESCENT

numRepetitions = 4 # FIX!!! repetitions of experiment due to stochastic nature

num_inputs = training_input.shape[1]
num_outputs = num_classes

idx_config = 0

for config in configurations:
  print("=======")
  print("Now running tests for config", config)

  learningRates = config['learningRates']
  num_hidden = config['hiddenLayerSizes']
  solver = config['solver']
  activation = config['activation']

  # Sequential network structure.
  model = Sequential()

  if len(num_hidden) == 0:
    print("Error: Must at least have one hidden layer!")
    sys.exit()

  # add first hidden layer connecting to input layer
  model.add(Dense(num_hidden[0], input_dim=num_inputs, activation=activation, ker

#  if dropout: # dropout at input layer is generally not recommended
#    # dropout of fraction dropout of the neurons and activation layer.
#    model.add(Dropout(dropout))
#  #  model.add(Activation("linear"))
```

```
if batch_normalization:
  model.add(BatchNormalization())

# potentially further hidden layers
for i in range(1, len(num_hidden)):
  # add hidden layer with len[i] neurons
  model.add(Dense(num_hidden[i], activation=activation, kernel_regularizer=regu
#  model.add(Activation("linear"))

  if dropout:
  # dropout of fraction dropout of the neurons and activation layer.
    model.add(Dropout(dropout))
  #  model.add(Activation("linear"))

  if batch_normalization:
    model.add(BatchNormalization())

# output layer
model.add(Dense(units=num_outputs, name = "output", kernel_regularizer=regulari

if dropout:
# dropout of fraction dropout of the neurons and activation layer.
  model.add(Dropout(dropout))
#  model.add(Activation("linear"))

# print configuration
print("\nModel configuration: ")
print(model.get_config())
print("\n")

# show how the model looks
model.summary()

optLearningRate = 0
optValidationAccuracy = 0

histories_lr = [] # remember history for each learning rate

for idx_lr in range(len(learningRates)):

  print("MODIFYING LEARNING RATE")
  learningRate = learningRates[idx_lr]
  if learningRateSchedule == True:
    lr_schedule = schedules.ExponentialDecay(initial_learning_rate = learningRa
    print("... applying exponential decay learning rate schedule with initial l
  else:
    lr_schedule = learningRate # constant learning rate
    print("... constant learning rate %f" % learningRate)

  train_loss = np.zeros(numRepetitions)
  train_acc = np.zeros(numRepetitions)
  val_loss = np.zeros(numRepetitions)
  val_acc = np.zeros(numRepetitions)
  test_loss = np.zeros(numRepetitions)
  test_acc = np.zeros(numRepetitions)
```

```
histories_rep = [] # (temporarily) remember history of each repetition
for idx_rep in range(numRepetitions):
  print("\nIteration %d..." % idx_rep)

  # compile model
  if solver == 'SGD':
    opt = SGD(learning_rate=lr_schedule, momentum=momentum, nesterov=nesterov
  elif solver == 'Adam':
    opt = Adam(learning_rate=lr_schedule)
  elif solver == 'Nadam':
    opt = Nadam(learning_rate=lr_schedule) # Nadam doesn't support adaptive l
  elif solver == 'Adadelta':
    opt = Adadelta(learning_rate=lr_schedule)
  elif solver == 'Adagrad':
    opt = Adagrad(learning_rate=lr_schedule)
  elif solver == 'RMSprop':
    opt = RMSprop(learning_rate=lr_schedule, momentum = momentum)
  model.compile(optimizer=opt,loss=tf.keras.losses.SparseCategoricalCrossentr

  # Training model with train data. Fixed random seed:
  num_epochs = 100 # FIX !!!
  batch_size = 1000 # FIX !!!
  history = model.fit(training_input, training_target, epochs=num_epochs, bat
  histories_rep.append(history) # remember all histories from all repetitions
  train_loss[idx_rep] = history.history['loss'][num_epochs-1]
  train_acc[idx_rep] = history.history['sparse_categorical_accuracy'][num_epo
  val_loss[idx_rep] = model.evaluate(validation_input, validation_target)[0]
  val_acc[idx_rep] = model.evaluate(validation_input, validation_target)[1]
  test_loss[idx_rep] = model.evaluate(test_input, test_target)[0]
  test_acc[idx_rep] = model.evaluate(test_input, test_target)[1]

# print results:
print("training loss (in brackets: mean +/- std):")
for i in range(numRepetitions):
    print("%f" % train_loss[i])
print("(%f +/- %f)\n" % (np.mean(train_loss), np.std(train_loss, ddof=1)))

print("training accuracy (in brackets: mean +/- std):")
for i in range(numRepetitions):
    print("%f" % train_acc[i])
print("(%f +/- %f)\n" % (np.mean(train_acc), np.std(train_acc, ddof=1)))

print("validation loss (in brackets: mean +/- std):")
for i in range(numRepetitions):
    print("%f" % val_loss[i])
print("(%f +/- %f)\n" % (np.mean(val_loss), np.std(val_loss, ddof=1)))

print("validation accuracy (in brackets: mean +/- std):")
for i in range(numRepetitions):
    print("%f" % val_acc[i])
print("(%f +/- %f)\n" % (np.mean(val_acc), np.std(val_acc, ddof=1)))

print("test loss (in brackets: mean +/- std):")
for i in range(numRepetitions):
    print("%f" % test_loss[i])
print("(%f +/- %f)\n" % (np.mean(test_loss), np.std(test_loss, ddof=1)))
```

```
    print("test accuracy (in brackets: mean +/- std):")
    for i in range(numRepetitions):
        print("%f" % test_acc[i])
    print("(%f +/- %f)\n" % (np.mean(test_acc), np.std(test_acc, ddof=1)))

    # remember history of best repetition (based on maximal validation accuracy)
    idx_best_rep = np.argmax(val_acc)

    # plot training loss and accuracy for best repetition
    print("\nbest repetition: experiment %d" % idx_best_rep)
    plt.plot(histories_rep[idx_best_rep].history['loss'], color = 'blue',
                 label = 'training loss')
    plt.plot(histories_rep[idx_best_rep].history['sparse_categorical_accuracy'],
                 label = 'traning accuracy')
    plt.xlabel('Epoch number')
    plt.ylim(0, 1)
    plt.legend()
    plt.show()

    # determine optimal learning rate (based on mean validation accuracy over rep
    if np.mean(val_acc) > optValidationAccuracy:
        optValidationAccuracy = np.mean(val_acc)
        opt_learning_rate[idx_config] = learningRate
        # remember history
        histories[idx_config] = histories_rep[idx_best_rep]
        # remember evaluation results
        final_training_loss[idx_config] = train_loss[idx_best_rep]
        final_training_accuracy[idx_config] = train_acc[idx_best_rep]
        final_validation_loss[idx_config] = val_loss[idx_best_rep]
        final_validation_accuracy[idx_config] = val_acc[idx_best_rep]
        final_test_loss[idx_config] = test_loss[idx_best_rep]
        final_test_accuracy[idx_config] = test_acc[idx_best_rep]

  print("\n\noptimal learning rate for this configuration: %f\n\n" % opt_learning

  # print evaluation results
  print("\nconfiguration %s:\n" % configurations[idx_config])
  print("optimal learning rate: %f" % opt_learning_rate[idx_config])
  print("final training loss: %f" % final_training_loss[idx_config])
  print("final training accuracy: %f" % final_training_accuracy[idx_config])
  print("final validation loss: %f" % final_validation_loss[idx_config])
  print("final validation accuracy: %f" % final_validation_accuracy[idx_config])
  print("final test loss: %f" % final_test_loss[idx_config])
  print("final test accuracy: %f" % final_test_accuracy[idx_config])

  # increment configuration index
  idx_config = idx_config + 1

###--------------------------------
# Summary: print evaluation results
###--------------------------------

print("\n\nSummary:\n\n")
for i in range(len(configurations)):
  print("\nconfiguration %s:\n" % configurations[i])
  print("optimal learning rate: %f" % opt_learning_rate[i])
  print("final training loss: %f" % final_training_loss[i])
```

```
  print("final training accuracy: %f" % final_training_accuracy[i])
  print("final validation loss: %f" % final_validation_loss[i])
  print("final validation accuracy: %f" % final_validation_accuracy[i])
  print("final test loss: %f" % final_test_loss[i])
  print("final test accuracy: %f" % final_test_accuracy[i])

###--------------------
# Summary: plot results
###--------------------

# plot setup
num_rows = np.int(np.ceil(len(configurations)/2))
fig, axes = plt.subplots(num_rows, 2, figsize=(15, 10))
fig.tight_layout() # improve spacing between subplots, doesn't work
plt.subplots_adjust(left=0.125, right=0.9, bottom=0.1, top=0.9, wspace=0.2, hspac
legend = []
i = 0
axes_indices = {}

if (len(configurations) <= 2):
  for i in range(len(configurations)):
    axes_indices[i] = i
else:
  for i in range(num_rows):
    axes_indices[2*i] = (i, 0)
    axes_indices[2*i+1] = (i, 1)

for i in range(len(configurations)):
  # plot loss
  axes[axes_indices[i]].set_title('configuration ' + str(i))
  if i == 8 or i == 9:
    axes[axes_indices[i]].set_xlabel('Epoch number')
  axes[axes_indices[i]].set_ylim(0, 1)
  axes[axes_indices[i]].plot(histories[i].history['loss'], color = 'blue',
              label = 'training loss')
  axes[axes_indices[i]].plot(histories[i].history['sparse_categorical_accuracy'],
              label = 'traning accuracy')
  axes[axes_indices[i]].legend()

  i = i + 1


# show the plot
plt.show()
```

Experiment 1

configuration {'learningRates': [0.1, 0.11], 'hiddenLayerSizes': [250, 150], 'solver': 'SGD', 'activation': 'relu'}:

optimal learning rate: 0.110000 final training loss: 4.589739 final training accuracy: 0.450040 final validation loss: 3.016404 final validation accuracy: 0.639700 final test loss: 3.021441 final test accuracy: 0.628000

configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [200, 80], 'solver': 'Adam', 'activation': 'relu'}:

optimal learning rate: 0.110000 final training loss: 22.865499 final training accuracy: 0.194080 final validation loss: 23.130457 final validation accuracy: 0.131200 final test loss: 23.136959 final test accuracy: 0.127600

configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [120, 70], 'solver': 'Nadam', 'activation': 'relu'}:

optimal learning rate: 0.120000 final training loss: 34.055054 final training accuracy: 0.123080 final validation loss: 29.317556 final validation accuracy: 0.134600 final test loss: 29.317846 final test accuracy: 0.130700

configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [200, 100], 'solver': 'Adagrad', 'activation': 'relu'}:

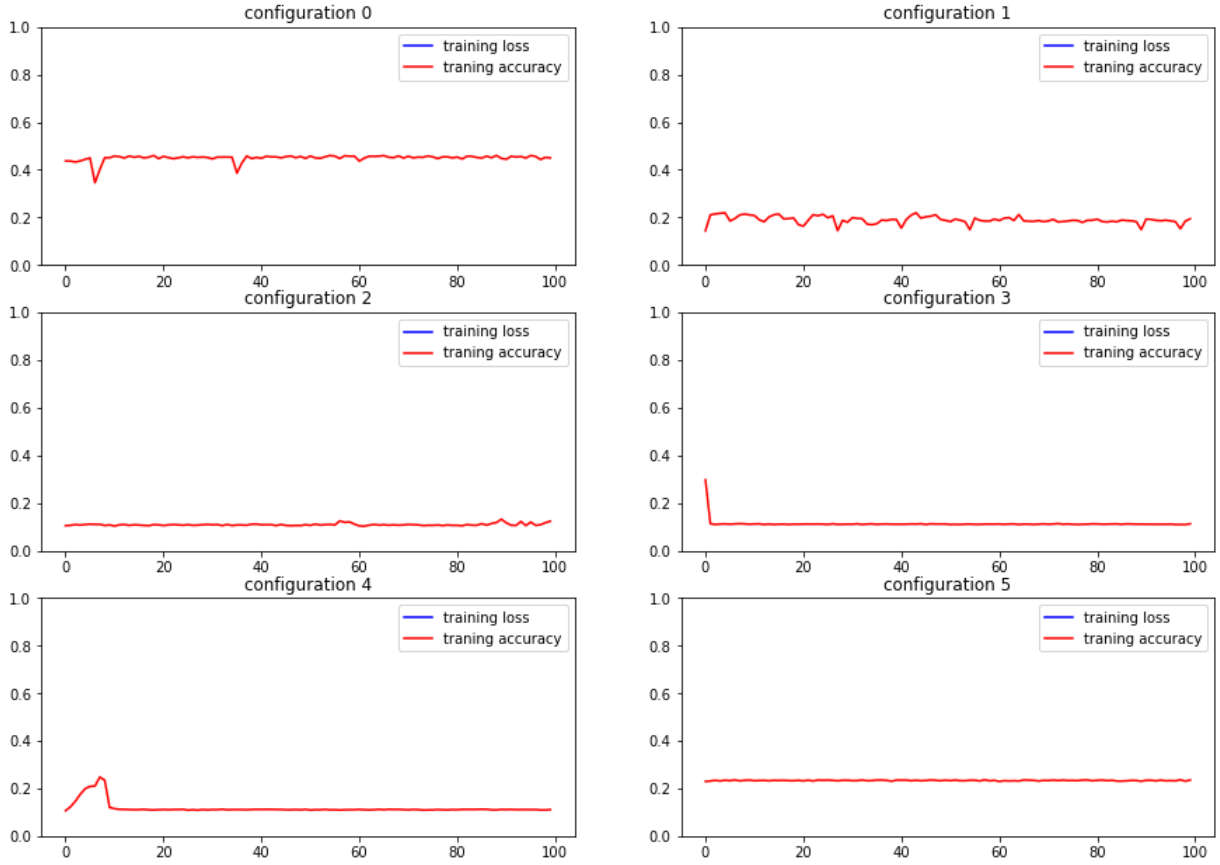
optimal learning rate: 0.100000 final training loss: 3.454568 final training accuracy: 0.112080 final validation loss: 3.453527 final validation accuracy: 0.106400 final test loss: 3.453347 final test accuracy: 0.113500

configuration {'learningRates': [0.1, 0.11], 'hiddenLayerSizes': [250, 150], 'solver': 'Adadelta', 'activation': 'relu'}:

optimal learning rate: 0.100000 final training loss: 2.877057 final training accuracy: 0.110820 final validation loss: 2.879114 final validation accuracy: 0.106400 final test loss: 2.878903 final test accuracy: 0.113500

configuration {'learningRates': [0.1, 0.11], 'hiddenLayerSizes': [170, 120], 'solver': 'RMSProp', 'activation': 'relu'}:

optimal learning rate: 0.110000 final training loss: 2.960142 final training accuracy: 0.235280 final validation loss: 2.851573 final validation accuracy: 0.267300 final test loss: 2.854919 final test accuracy: 0.268600



Experiment 2

configuration {'learningRates': [0.11, 0.13], 'hiddenLayerSizes': [60, 30], 'solver': 'SGD', 'activation': 'relu'}:

optimal learning rate: 0.130000 final training loss: 1.120146 final training accuracy: 0.772360 final validation loss: 0.790587 final validation accuracy: 0.931100 final test loss: 0.798023 final test accuracy: 0.926100

configuration {'learningRates': [0.1, 0.11], 'hiddenLayerSizes': [60, 30], 'solver': 'Adam', 'activation': 'relu'}:

optimal learning rate: 0.100000 final training loss: 3.831295 final training accuracy: 0.134260 final validation loss: 3.213572 final validation accuracy: 0.191100 final test loss: 3.211654 final test accuracy: 0.196000

configuration {'learningRates': [0.11, 0.13], 'hiddenLayerSizes': [70, 50], 'solver': 'Nadam', 'activation': 'relu'}:

optimal learning rate: 0.110000 final training loss: 12.635813 final training accuracy: 0.112240 final validation loss: 10.809395 final validation accuracy: 0.239000 final test loss: 10.809360 final test accuracy: 0.240300

configuration {'learningRates': [0.11, 0.13], 'hiddenLayerSizes': [70, 60], 'solver': 'Adagrad', 'activation': 'relu'}:

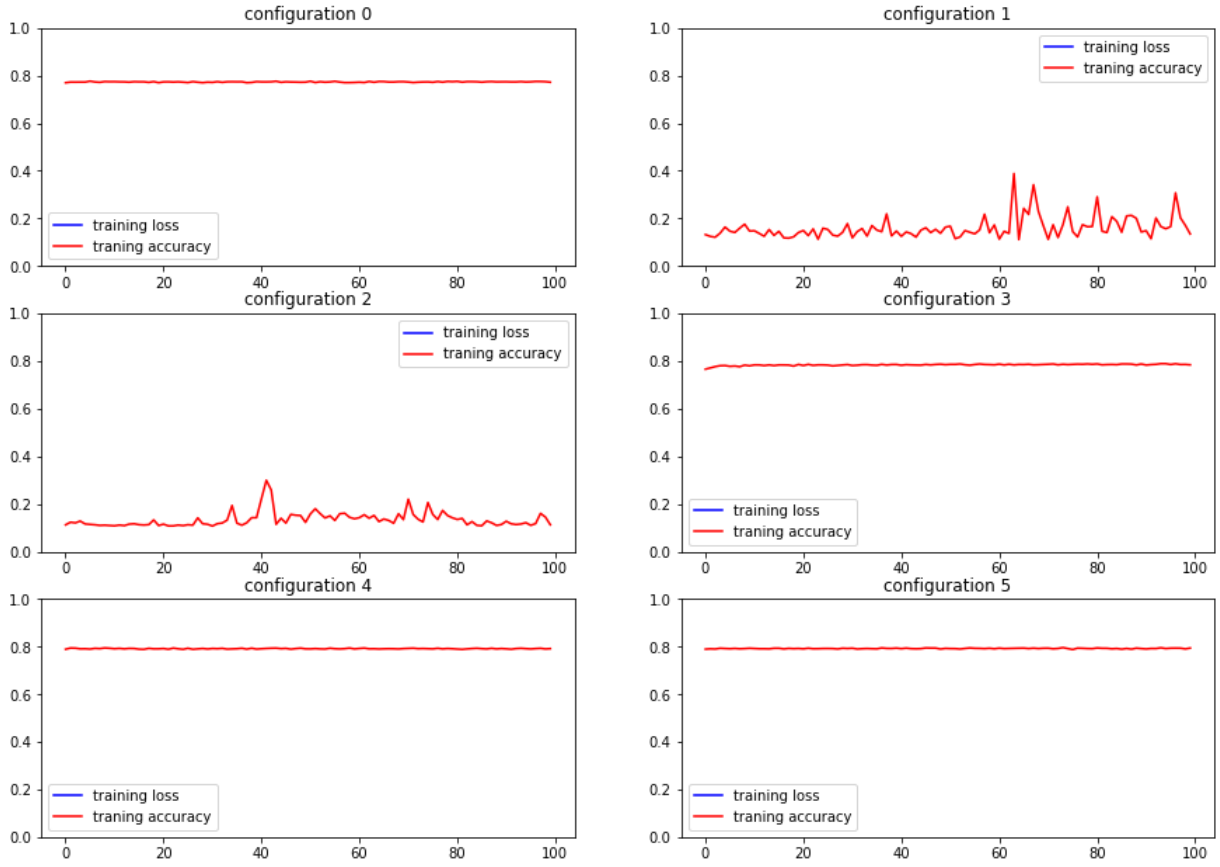
optimal learning rate: 0.110000 final training loss: 1.071748 final training accuracy: 0.784500 final validation loss: 0.775677 final validation accuracy: 0.931300 final test loss: 0.782049 final test accuracy: 0.924900

configuration {'learningRates': [0.11, 0.13], 'hiddenLayerSizes': [70, 70], 'solver': 'Adadelta', 'activation': 'relu'}:

optimal learning rate: 0.130000 final training loss: 1.051697 final training accuracy: 0.791740 final validation loss: 0.767068 final validation accuracy: 0.933200 final test loss: 0.772356 final test accuracy: 0.927500

configuration {'learningRates': [0.11, 0.13], 'hiddenLayerSizes': [70, 80], 'solver': 'RMSProp', 'activation': 'relu'}:

optimal learning rate: 0.130000 final training loss: 1.045929 final training accuracy: 0.794160 final validation loss: 0.762948 final validation accuracy: 0.932400 final test loss: 0.769738 final test accuracy: 0.926900



Experiement 3

configuration {'learningRates': [0.2, 0.25], 'hiddenLayerSizes': [220], 'solver': 'Adam', 'activation': 'relu'}:

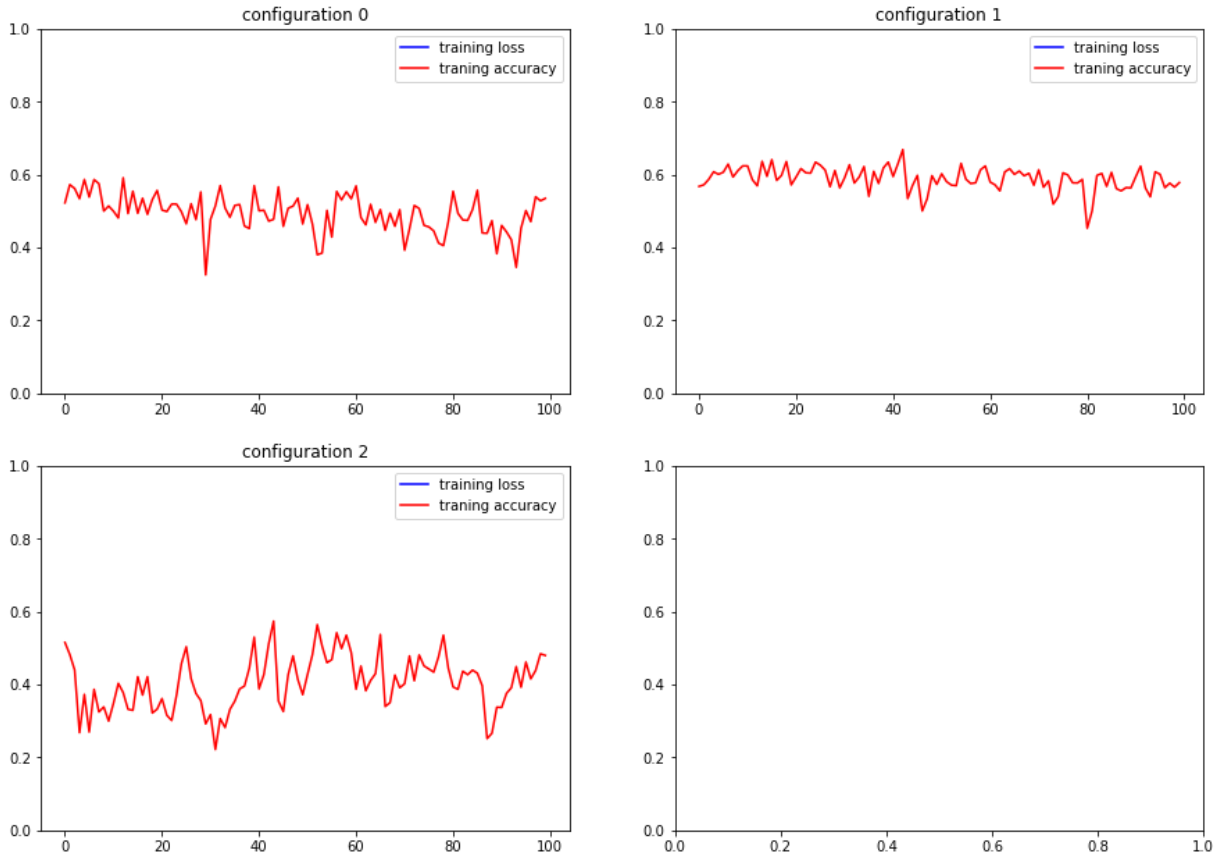
optimal learning rate: 0.250000 final training loss: 214.462936 final training accuracy: 0.534420 final validation loss: 241.456207 final validation accuracy: 0.701900 final test loss: 241.500000 final test accuracy: 0.697600

configuration {'learningRates': [0.2, 0.25], 'hiddenLayerSizes': [220], 'solver': 'Nadam', 'activation': 'relu'}:

optimal learning rate: 0.250000 final training loss: 292.786682 final training accuracy: 0.577840 final validation loss: 283.038818 final validation accuracy: 0.807900 final test loss: 283.473114 final test accuracy: 0.790300

configuration {'learningRates': [0.11, 0.13], 'hiddenLayerSizes': [220], 'solver': 'RMSProp', 'activation': 'relu'}:

optimal learning rate: 0.130000 final training loss: 325.904144 final training accuracy: 0.479960 final validation loss: 278.261871 final validation accuracy: 0.707900 final test loss: 278.766479 final test accuracy: 0.694100



After few experiments it became evident that the Adam and Nadam performs well for l1 regularization and for single hidden layer but works bad for multi hidden layer and for the other optimizers l2 regularization worked well.

Exercise 4 (Vanishing gradient)

a) The Jupyter notebook implements a multi-layer perceptron for use on the MNIST digit classification problem. Apart from the training loss and accuracy, it also displays a histogram of the weights (between the input and the first hidden layer) after initialization and at the end of the training, and visualizes the weights (between the input layer and 16 hidden neurons of the first

hidden layer). Using a sigmoid activation function, compare the output for a single hidden layer, five and six hidden layers. Then change to a ReLU activation function and inspect the results for six hidden layers. Discuss your findings.

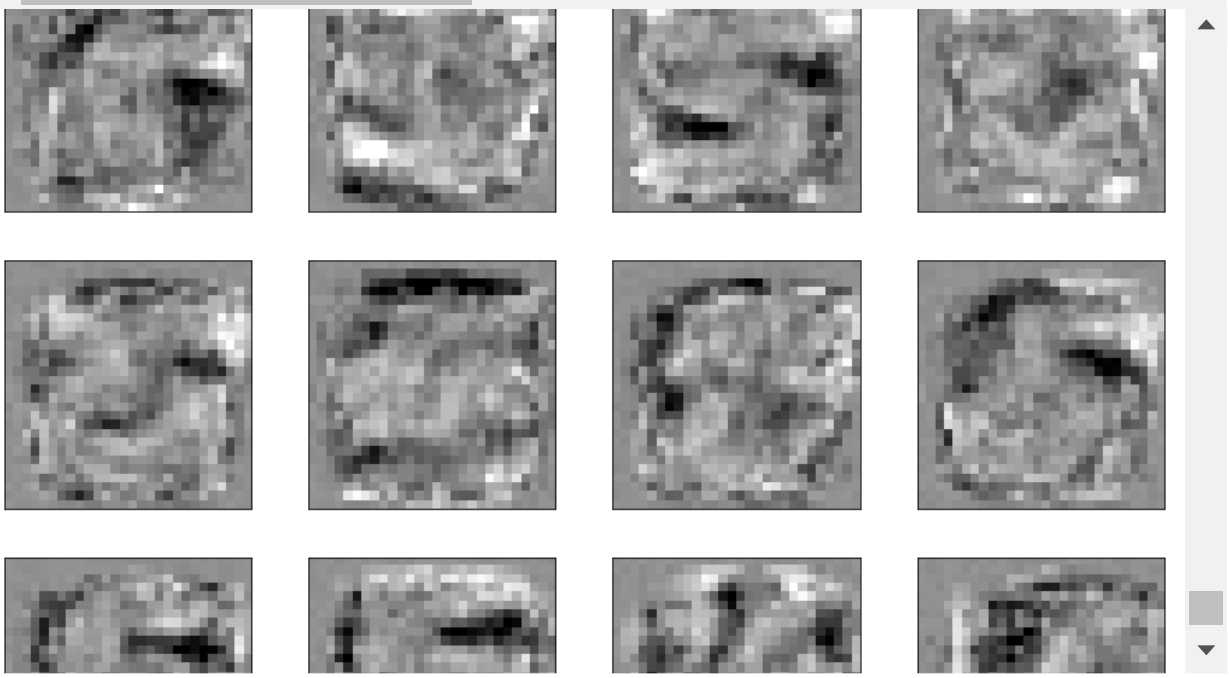
In [6]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from os.path import join
from tensorflow.keras.layers import Dense, Dropout, Activation, BatchNormalizatic
from tensorflow.keras import Model, Input, Sequential
from tensorflow.keras.optimizers import SGD, Adam, Adadelta, Adagrad, Nadam, RMSp
from tensorflow.keras.utils import normalize
import tensorflow.keras.datasets as tfds
import tensorflow.keras.initializers as tfi
import tensorflow.keras.regularizers as tfr

###--------
# load data
###--------

(training_input, training_target), (test_input, test_target)  = tfds.mnist.load_d

# Reserve 10,000 samples for validation
validation_input = training_input[-10000:]
validation_target = training_target[-10000:]
training_input = training_input[:-10000]
training_target = training_target[:-10000]

print("training input shape: %s, training target shape: %s"  % (training_input.sh
print("validation input shape: %s, validation target shape: %s"  % (validation_in
print("test input shape: %s, test target shape: %s"  % (test_input.shape, test_ta
# range of input values: 0 ... 255
print("\n")

###-----------
# process data
###-----------

# Note: shuffling is performed in fit method

# scaling inputs from range 0 ... 255 to range [0,1] if desired
scale_inputs = True # scale inputs to range [0,1]
if scale_inputs:
  training_input = training_input / 255
  validation_input = validation_input / 255
  test_input = test_input / 255

# flatten inputs to vectors
training_input = training_input.reshape(training_input.shape[0], training_input.s
validation_input = validation_input.reshape(validation_input.shape[0], validatior
test_input = test_input.reshape(test_input.shape[0], test_input.shape[1] * test_i
print(training_input.shape)
print(validation_input.shape)
print(test_input.shape)

num_classes = 10 # 10 digits

###-----------
# define model
###-----------
```

```
num_inputs = training_input.shape[1]
num_hidden = [100,80,70,60,50,40] # FIX!!!
num_outputs = num_classes

initialLearningRate = 0.01 # FIX!!!
# select constant learning rate or (flexible) learning rate schedule,
# i.e. select one of the following two alternatives
lr_schedule = initialLearningRate # constant learning rate
# lr_schedule = schedules.ExponentialDecay(initial_learning_rate = initialLearnir

solver = 'RMSprop'
activation = 'sigmoid' # FIX!!! e.g. sigmoid or relu
dropout = 0 # 0 if no dropout, else fraction of dropout units (e.g. 0.2)   # FIX!
batch_normalization = False

weight_init = tfi.glorot_uniform() # FIX!!! default: glorot_uniform(); e.g. gloro
bias_init = tfi.Zeros() # FIX!!! default: Zeros(); for some possible values see w

regularization_weight = 0.0 # 0 for no regularization or e.g. 0.01 to apply regul
regularizer = tfr.l1(l=regularization_weight) # or l2 or l1_l2; used for both wei

num_epochs = 50 # FIX !!!
batch_size = 1000 # FIX !!!

# Sequential network structure.
model = Sequential()

if len(num_hidden) == 0:
  print("Error: Must at least have one hidden layer!")
  sys.exit()

# add first hidden layer connecting to input layer

model.add(Dense(num_hidden[0], input_dim=num_inputs, activation=activation, kerne

# if dropout: # dropout at input layer is generally not recommended
#  # dropout of fraction dropout of the neurons and activation layer.
#  model.add(Dropout(dropout))
# #  model.add(Activation("linear"))

if batch_normalization:
  model.add(BatchNormalization())

# potentially further hidden layers
for i in range(1, len(num_hidden)):
  # add hidden layer with len[i] neurons
  model.add(Dense(num_hidden[i], activation=activation, kernel_initializer=weight
#  model.add(Activation("linear"))

  if dropout:
  # dropout of fraction dropout of the neurons and activation layer.
    model.add(Dropout(dropout))
  #  model.add(Activation("linear"))

  if batch_normalization:
    model.add(BatchNormalization())
```

```
# output layer
model.add(Dense(units=num_outputs, name = "output", kernel_initializer=weight_ini

if dropout:
# dropout of fraction dropout of the neurons and activation layer.
  model.add(Dropout(dropout))
#  model.add(Activation("linear"))

# print configuration
print("\nModel configuration: ")
print(model.get_config())
print("\n")
print("... number of layers: %d" % len(model.layers))

# show how the model looks
model.summary()

# compile model
if solver == 'SGD':
  momentum = 0 # e.g. 0.0, 0.5, 0.9 or 0.99
  nesterov = False
  opt = SGD(learning_rate=lr_schedule, momentum=momentum, nesterov=nesterov) # SG
elif solver == 'Adam':
  opt = Adam(learning_rate=lr_schedule)
elif solver == 'Nadam':
  opt = Adam(learning_rate=lr_schedule)
elif solver == 'Adadelta':
  opt = Adam(learning_rate=lr_schedule)
elif solver == 'Adagrad':
  opt = Adam(learning_rate=lr_schedule)
elif solver == 'RMSprop':
  opt = RMSprop(learning_rate=lr_schedule)
model.compile(optimizer=opt,loss=tf.keras.losses.SparseCategoricalCrossentropy(fr

# histogram of weights (first layer) after initialization
weights = model.layers[0].get_weights()[0]
biases = model.layers[0].get_weights()[1]

nBins = 100
fig, axes = plt.subplots(1, 2, figsize=(15,10))
axes[0].hist(weights.flatten(), nBins)
axes[0].set_xlabel("weights")
axes[0].set_ylabel("counts")
axes[0].set_title("weight histogram after initialization")

axes[1].hist(biases.flatten(), nBins)
axes[1].set_xlabel("biases")
axes[1].set_ylabel("counts")
axes[1].set_title("bias histogram after initialization")
plt.show()

# visualize the weights between input layer and some
# of the hidden neurons of the first hidden layer after initialization
# model.layers[0].get_weights()[0] is a (784 x numHiddenNeurons) array
# model.layers[0].get_weights()[0].T (transpose) is a (numHiddenNeurons x 784) ar
# the first entry of which contains the weights of all inputs connecting
```

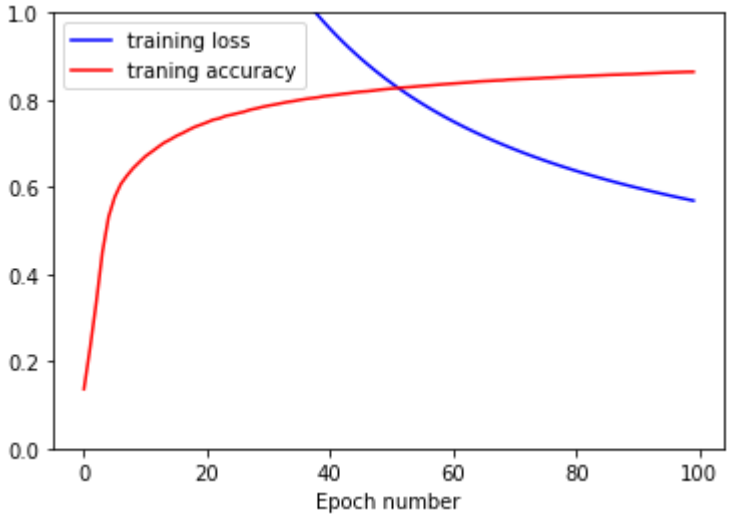
```
# to the first hidden neuron; those weights will be displayed in (28 x 28) format
# until all plots (4 x 4, i.e. 16) are "filled" or no more hidden neurons are lef
print("Visualization of the weights between input and some of the hidden neurons
fig, axes = plt.subplots(4, 4, figsize=(15,15))
# use global min / max to ensure all weights are shown on the same scale
weights = model.layers[0].get_weights()[0]
vmin, vmax = weights.min(), weights.max()
for coef, ax in zip(weights.T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
               vmax=.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())

plt.show()

# Training
history = model.fit(training_input, training_target, epochs=num_epochs, batch_siz

# plot training loss and accuracy
plt.plot(history.history['loss'], color = 'blue', label = 'training loss')
plt.plot(history.history['sparse_categorical_accuracy'], color = 'red', label = '
plt.xlabel('Epoch number')
plt.ylim(0, 1)
plt.legend()
plt.show()

# model evaluation
train_loss = history.history['loss'][num_epochs-1]
train_acc = history.history['sparse_categorical_accuracy'][num_epochs-1]
val_loss = model.evaluate(validation_input, validation_target)[0]
val_acc = model.evaluate(validation_input, validation_target)[1]
test_loss = model.evaluate(test_input, test_target)[0]
test_acc = model.evaluate(test_input, test_target)[1]

print("\n")
print("final training loss: %f" % train_loss)
print("final training accuracy: %f" % train_acc)
print("final validation loss: %f" % val_loss)
print("final validation accuracy: %f" % val_acc)
print("final test loss: %f" % test_loss)
print("final test accuracy: %f" % test_acc)
print("\n")

# histogram of weights (first layer) after training
weights = model.layers[0].get_weights()[0]
biases = model.layers[0].get_weights()[1]

nBins = 100
fig, axes = plt.subplots(1, 2, figsize=(15,10))
axes[0].hist(weights.flatten(), nBins)
axes[0].set_xlabel("weights")
axes[0].set_ylabel("counts")
axes[0].set_title("weight histogram after training")

axes[1].hist(biases.flatten(), nBins)
axes[1].set_xlabel("biases")
axes[1].set_ylabel("counts")
```

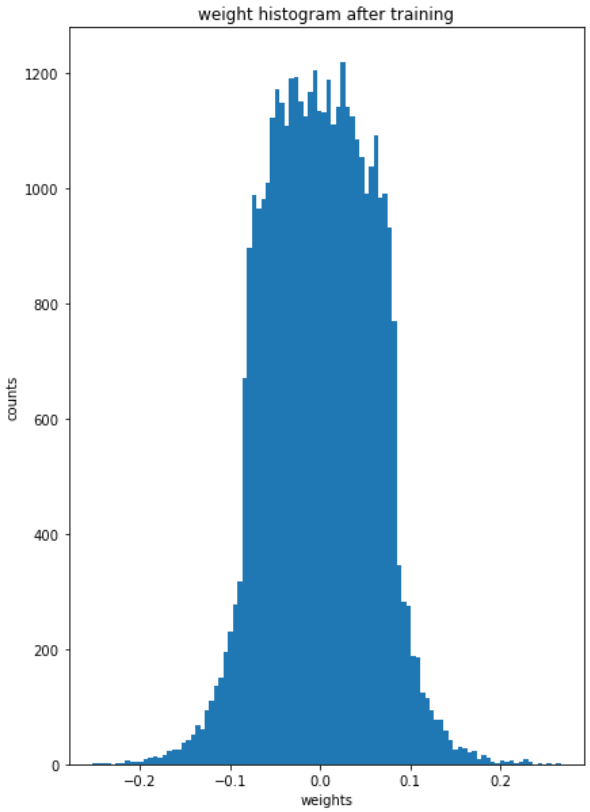
```
axes[1].set_title("bias histogram after training")
plt.show()

# visualize the weights between input layer and some
# of the hidden neurons of the first hidden layer after training
# model.layers[0].get_weights()[0] is a (784 x numHiddenNeurons) array
# model.layers[0].get_weights()[0].T (transpose) is a (numHiddenNeurons x 784) ar
# the first entry of which contains the weights of all inputs connecting
# to the first hidden neuron; those weights will be displayed in (28 x 28) format
# until all plots (4 x 4, i.e. 16) are "filled" or no more hidden neurons are lef
print("Visualization of the weights between input and some of the hidden neurons
fig, axes = plt.subplots(4, 4, figsize=(15,15))
# use global min / max to ensure all weights are shown on the same scale
weights = model.layers[0].get_weights()[0]
vmin, vmax = weights.min(), weights.max()
for coef, ax in zip(weights.T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
               vmax=.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())

plt.show()
```

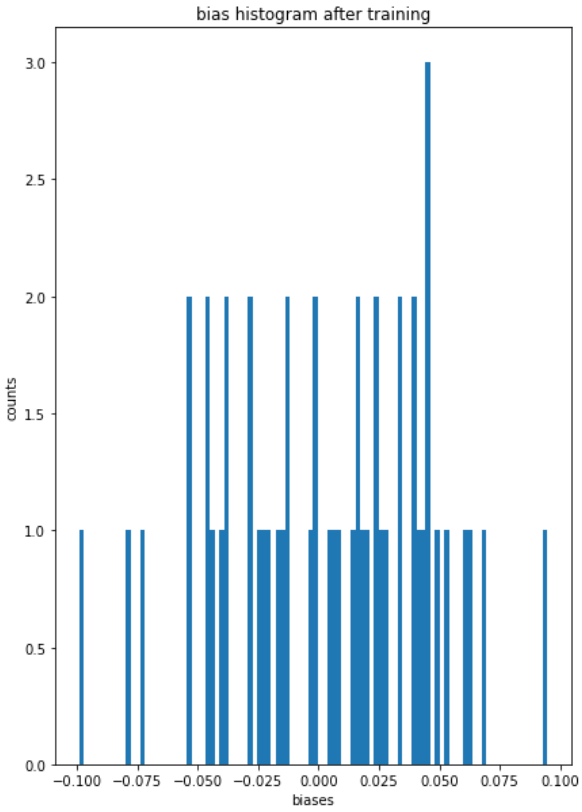


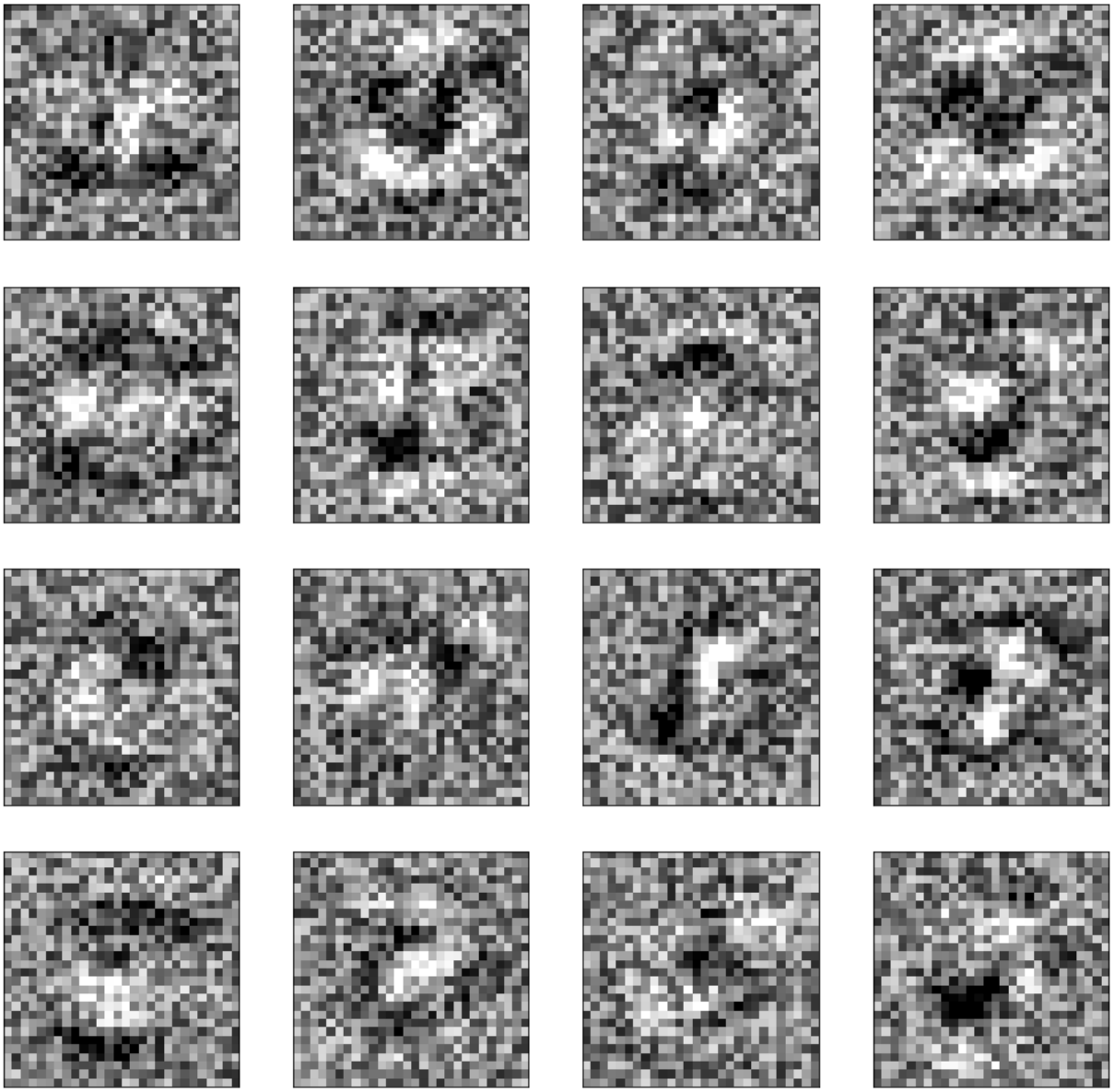
Single Layer sigmoid

final training loss: 0.568782 final training accuracy: 0.864380 final validation loss: 0.526777 final validation accuracy: 0.882300 final test loss: 0.542283 final test accuracy: 0.875700


training loss
traning accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Epoch number

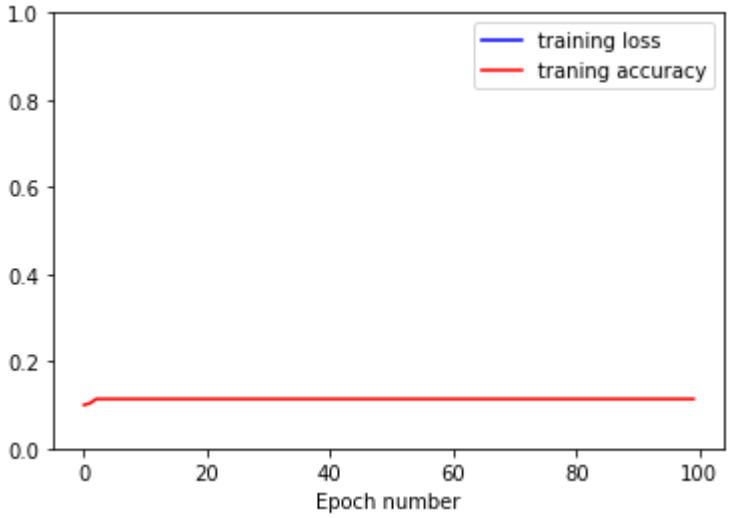

weight histogram after training
counts
weights
1200
1000
800
600
400
200
0
−0.2
−0.1
0.0
0.1
0.2

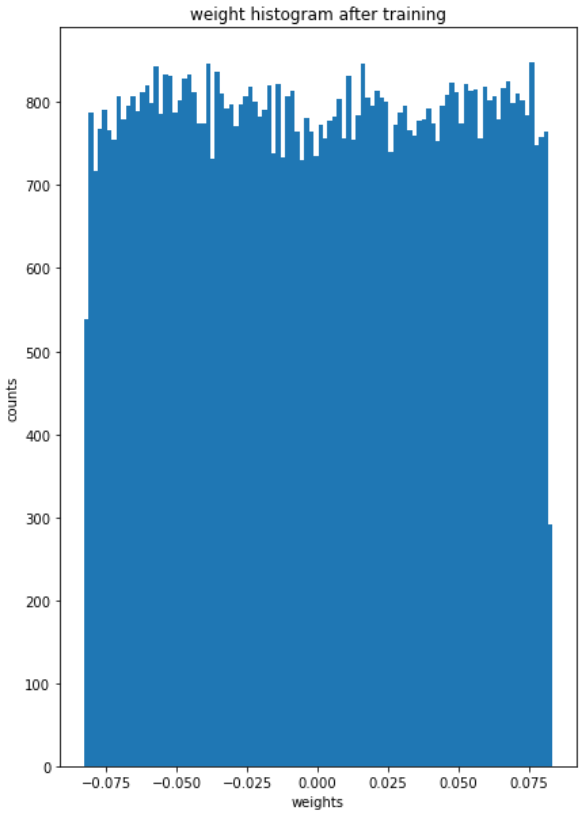

bias histogram after training
counts
biases
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.100
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075
0.100

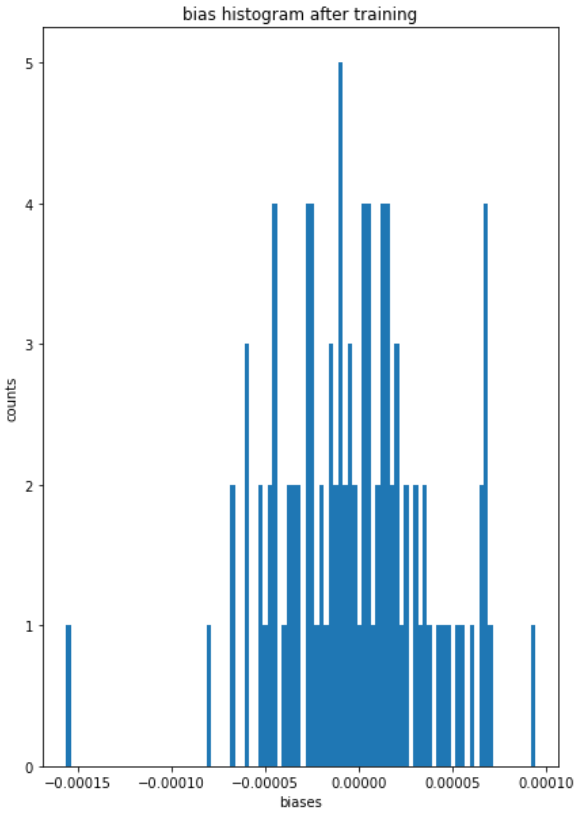


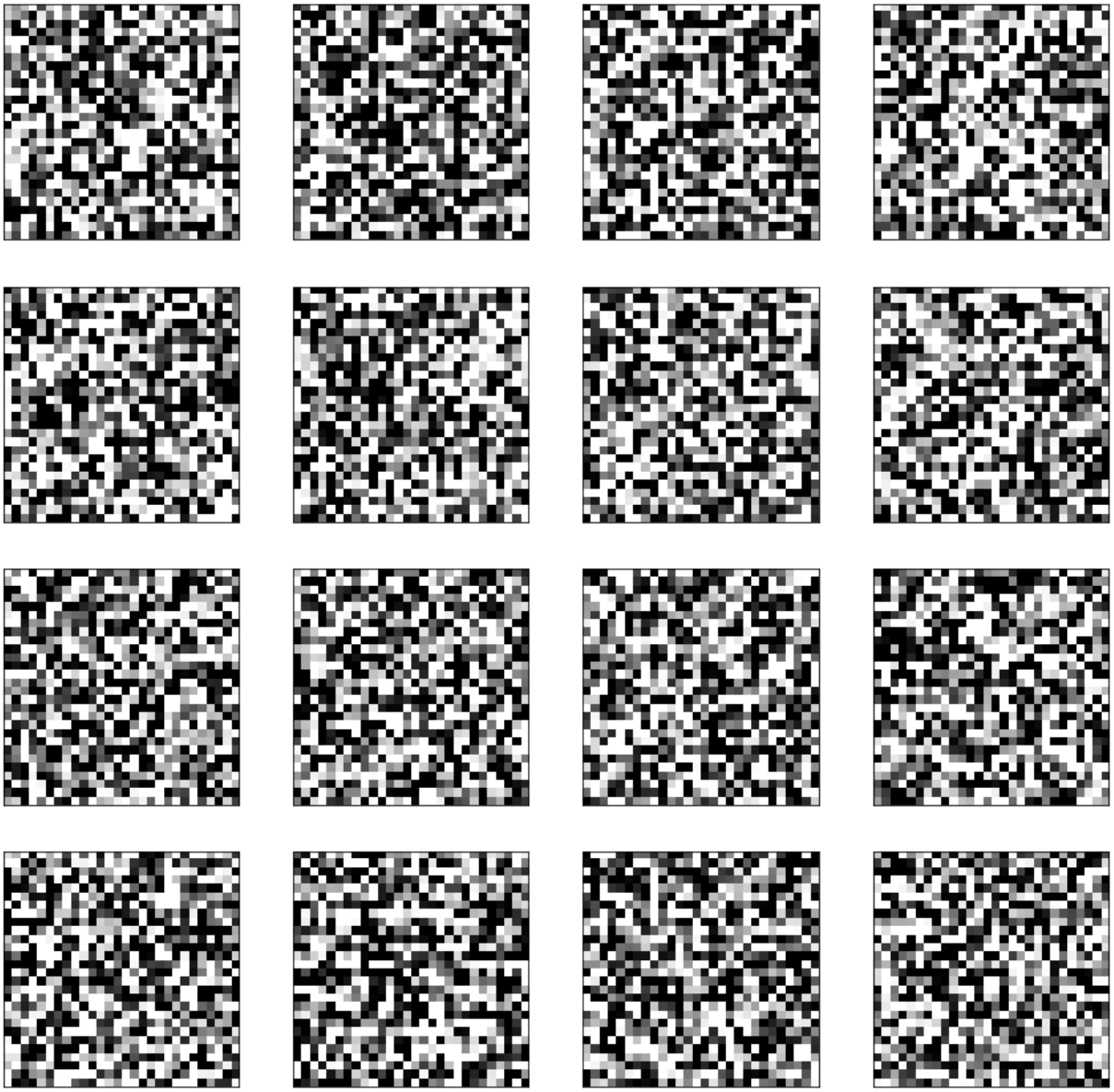
Five Layers sigmoid

final training loss: 2.300934 final training accuracy: 0.113560 final validation loss: 2.301861 final validation accuracy: 0.106400 final test loss: 2.300892 final test accuracy: 0.113500


training loss
traning accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Epoch number

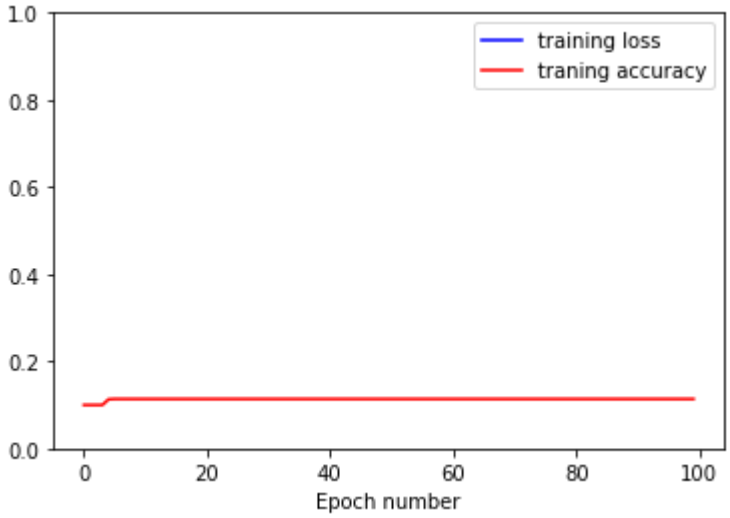

weight histogram after training
counts
weights
800
700
600
500
400
300
200
100
0
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075

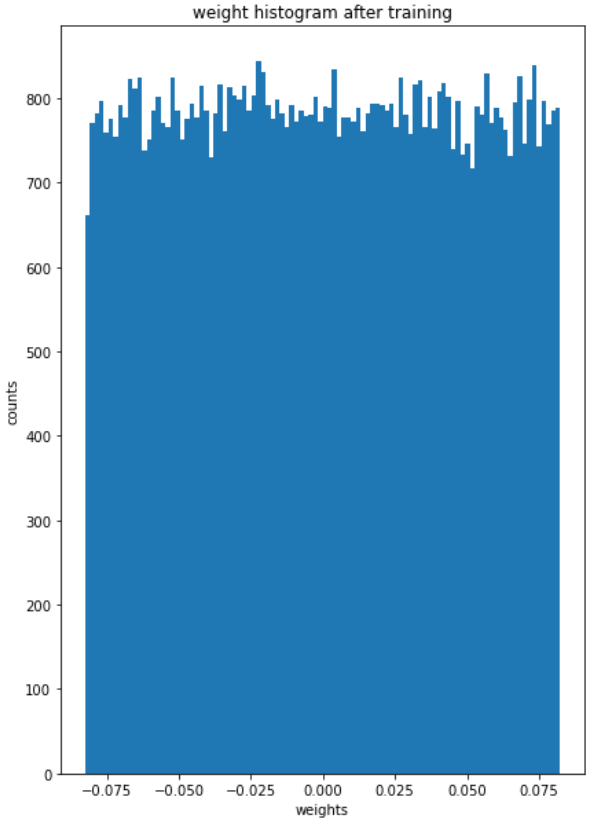

bias histogram after training
counts
biases
5
4
3
2
1
0
−0.00015
−0.00010
−0.00005
0.00000
0.00005
0.00010

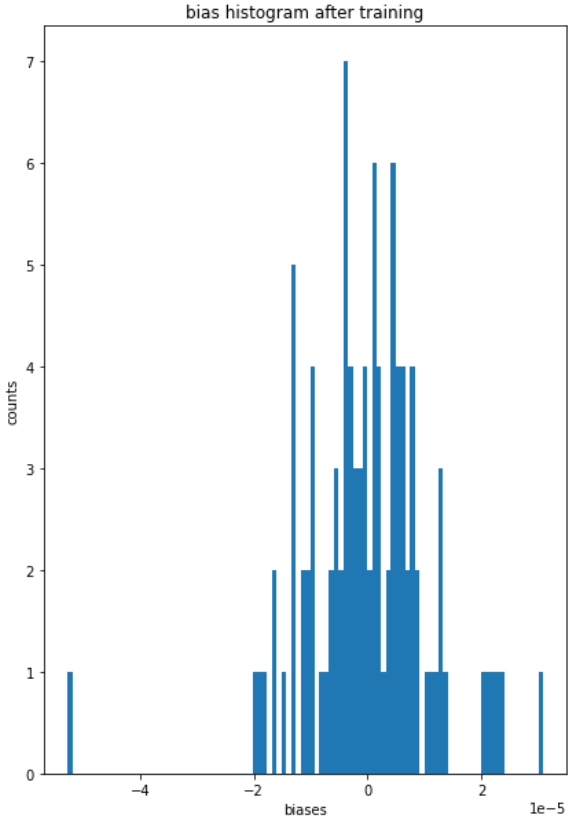


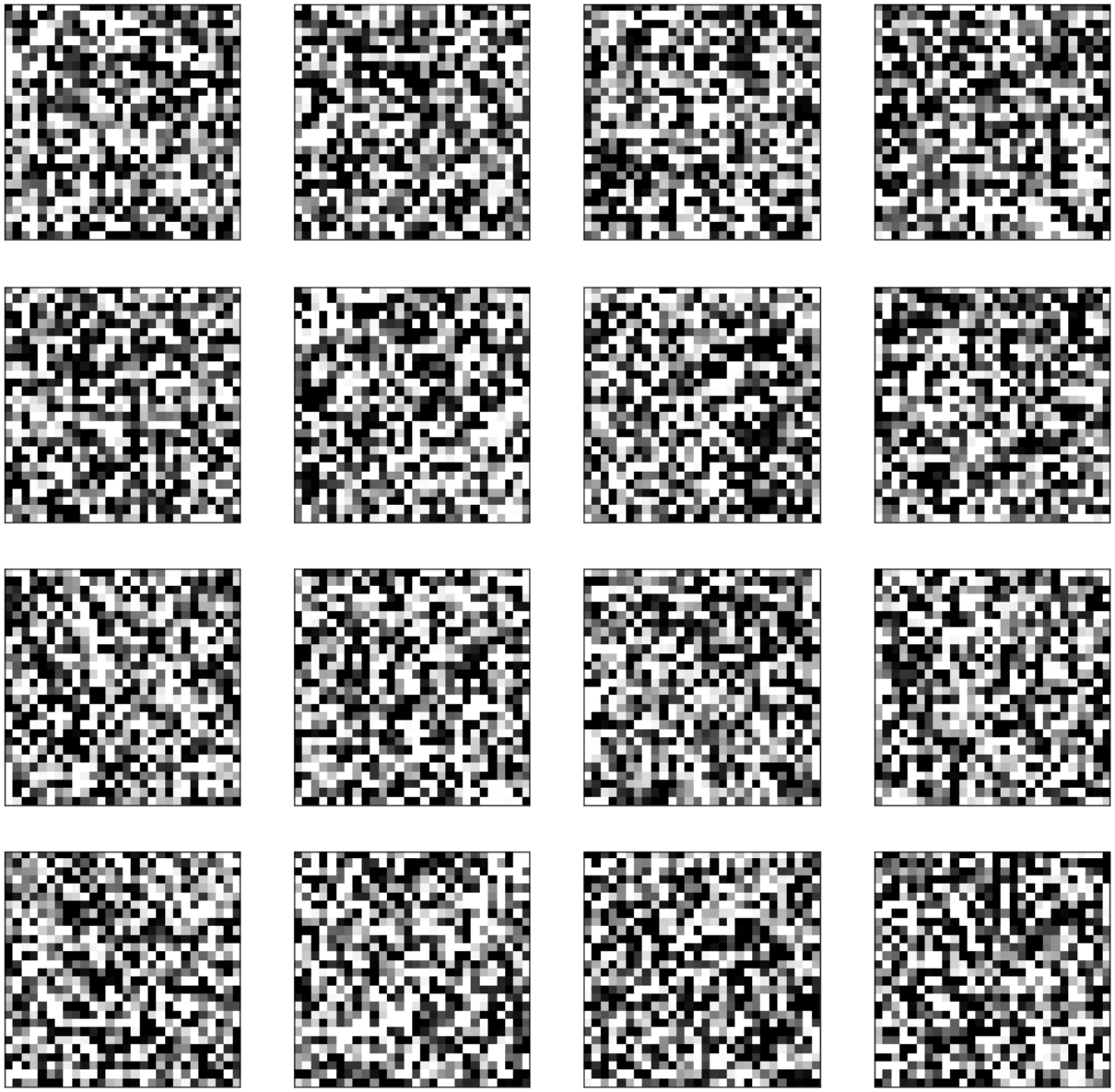
Six Layers sigmoid

final training loss: 2.301056 final training accuracy: 0.113560 final validation loss: 2.302010 final validation accuracy: 0.106400 final test loss: 2.301011 final test accuracy: 0.113500


training loss
traning accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Epoch number

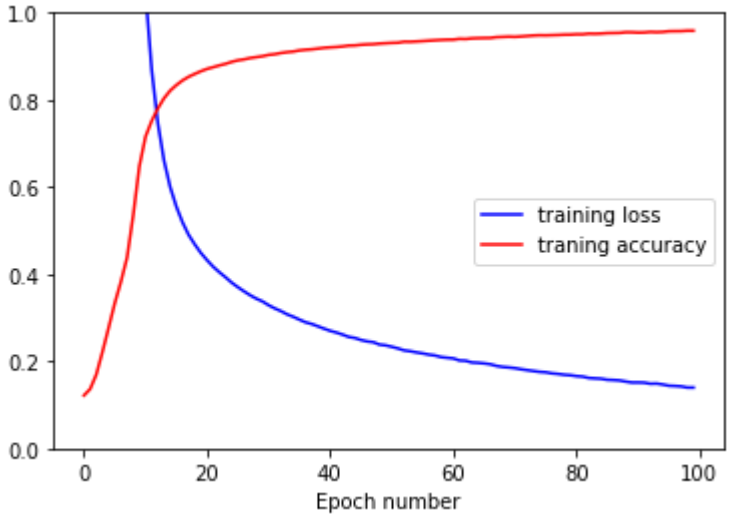

weight histogram after training
counts
weights
−0.075
−0.050
−0.025
0.000
0.025
0.050
0.075

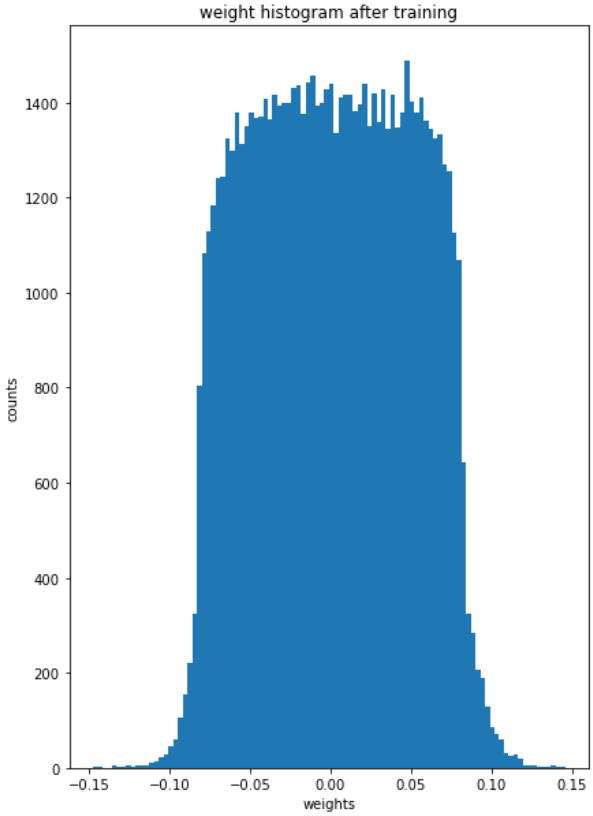

bias histogram after training
counts
biases
1e−5

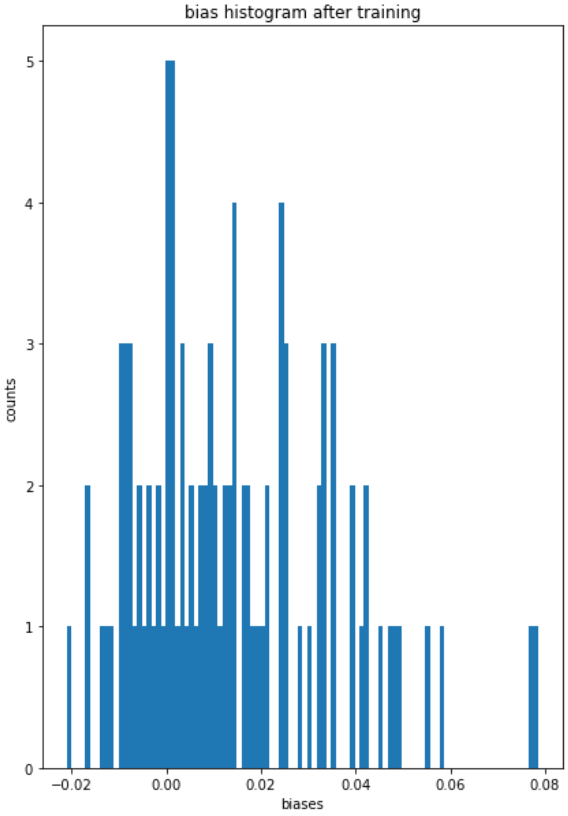


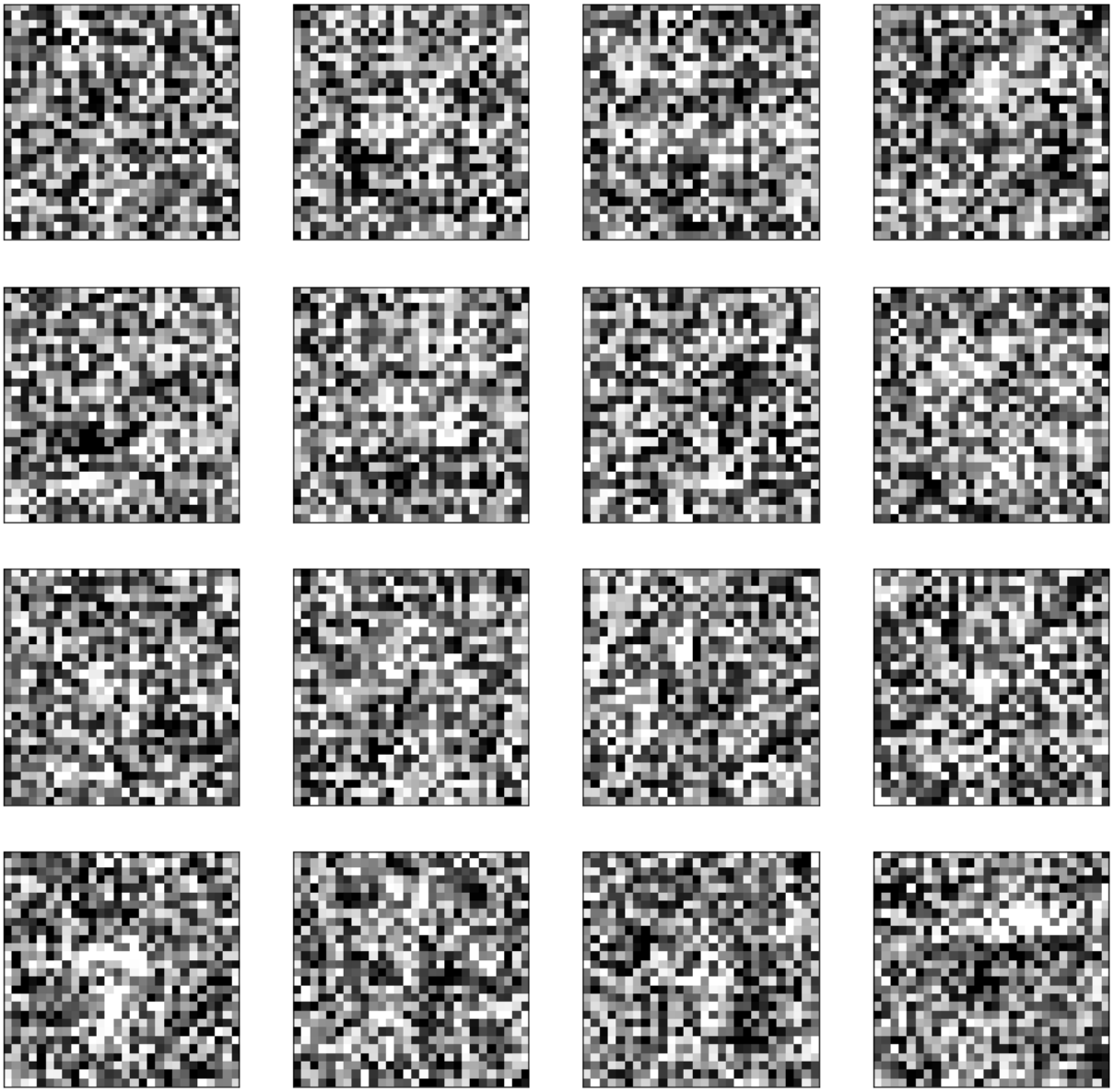
Six Layers ReLU

final training loss: 0.139757 final training accuracy: 0.958740 final validation loss: 0.154721 final validation accuracy: 0.956300 final test loss: 0.156863 final test accuracy: 0.953100


1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Epoch number
training loss
traning accuracy


weight histogram after training
counts
weights
1400
1200
1000
800
600
400
200
0
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15


bias histogram after training
counts
biases
5
4
3
2
1
0
−0.02
0.00
0.02
0.04
0.06
0.08



sigmoid works well for SGD optimizer and the single hidden layer NN but for the five and six layers its accuracy dropped because of the vanisihing gradient but for relu, six hidden layers NN works well with SGD optimizer.

b) Give a theoretical justification, why the weights and biases of neurons in the first hidden layers in a multi-layer perceptron with many hidden layers are modified only slowly when using a sigmoid activation function and gradient descent. To this end, consider – as an example – a simplified network with three hidden layers (and a single neuron per layer), compute and analyse the change of the bias of the first hidden neuron with respect to a change in the cost function C. What changes in your analysis when using a ReLU activation function instead of a sigmoid?

In [ ]:
```
Answer: Write your answer here.
```

c) Starting from your analysis for the multi-layer perceptron with six hidden layers and sigmoid activation function in part a), try to find other model configurations which lead to a successful training. You may modify e.g. the learning rate and batch size, the weight and bias initialization,

apply batch normalization and / or dropout, and add regularization.

configuration: solver=RMSprop, hidden layer=6, epoch=50, batch size = 1000

number of neuorns in layers = [100,80,70,60,50,40]

final training loss: 0.013187 final training accuracy: 0.996920 final validation loss: 0.174895 final validation accuracy: 0.972700 final test loss: 0.173825 final test accuracy: 0.973300