**Answer Sheet: Lab4 , Team : 15**


| Faiz Ahmed | Mithun Das | Mutasim Fuad Ansari | Mohammad Abir Reza |
|---|---|---|---|
| 1152231 | 1151651 | 1152109 | 1151705 |
| stu225473@mail.uni-kiel.de | stu225039@mail.uni-kiel.de | stu225365@mail.uni-kiel.de | stu225093@mail.uni-kiel.de |


# 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

# Answer

## Loss function:

Loss is nothing but a prediction error of Neural Net. And the method to calculate the loss is called Loss Function.There are few common loss fuctions tat are used such as,Mean Squared Error (MSE), cross-entropy loss function,Log-likelihood loss for softmax activation function.

## Stochastic gradient descent :

Gradient descent is an iterative algorithm for optimization, that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function. But, in large data set with redundant data, gradient descent is not efficient. Because it increases more computational work. SGD randomly picks single training example per epoch.

## Mini-batch :

In mini-batch, small portion of training data per epoch is used. Thumb rule for selecting the size of mini-batch is in power of 2 like 32, 64, 128 .

## Regularization :

Regularization is the idea of restricting the degrees(reducing weights or hidden-layer) of fredom of a machine learning model for the perpous of more efficient learning.

## Dropout :

Dropout is a regularization technique where some nodes remains inactive in hidden-layer randomly for reducing complexity and increasing efficiency.

## Batch normalization :

To improve accuracy and speed up training Batch normalization is applied. Batch normalization (BN) is a technique to normalize activations in intermediate layers of deep neural networks.

## Learning with momentum :

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 too large, the weight update may overshoot, leading to an oscillating loss function. In stochastic gradient descent the true gradient of the loss function is approximated by the average gradient calculated on a small mini-batch of training examples. Thus, the weight changes will not be perpendicular to the isocontours of the loss function, and take different directions at each weight update step. If the learning rate is small enough, this erratic behavior of the weight updates will still lead to the local minimum of the loss function. Learning with momentum is a compromise that smoothes the erratic behavior of the mini-batch updates, without slowing down the learning too much.

## Data augmentation :

Data augmentation is the process of artificially generating additional training data with more variation by rotation,scale and some other methods, by applying some distortion to original training images.

## 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 a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks. Deep learning use multiple layers to progressively extract higher level features from raw input.

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.

# Answer

## i. Binary Step Function:

a. Mathematical Equation: f(x) = 0 if x>= 0

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

## ii. Linear Function:

a. Mathematical Equation: f(x) = ax

b. Use case: It is used for Regression related tasks.

## iii. Sigmoid Function:

a. Mathematical Equation: f(x)=1/(1+e^-x)

b. Use case: classify the values to particular classes.

## iv. Tanh:

a. Mathematical Equation: tanh(x)=2/(1+e^(-2x)) -1

b. Use case: Classification.

## v. ReLU:

a. Mathematical Equation: f(x)=max(0,x)

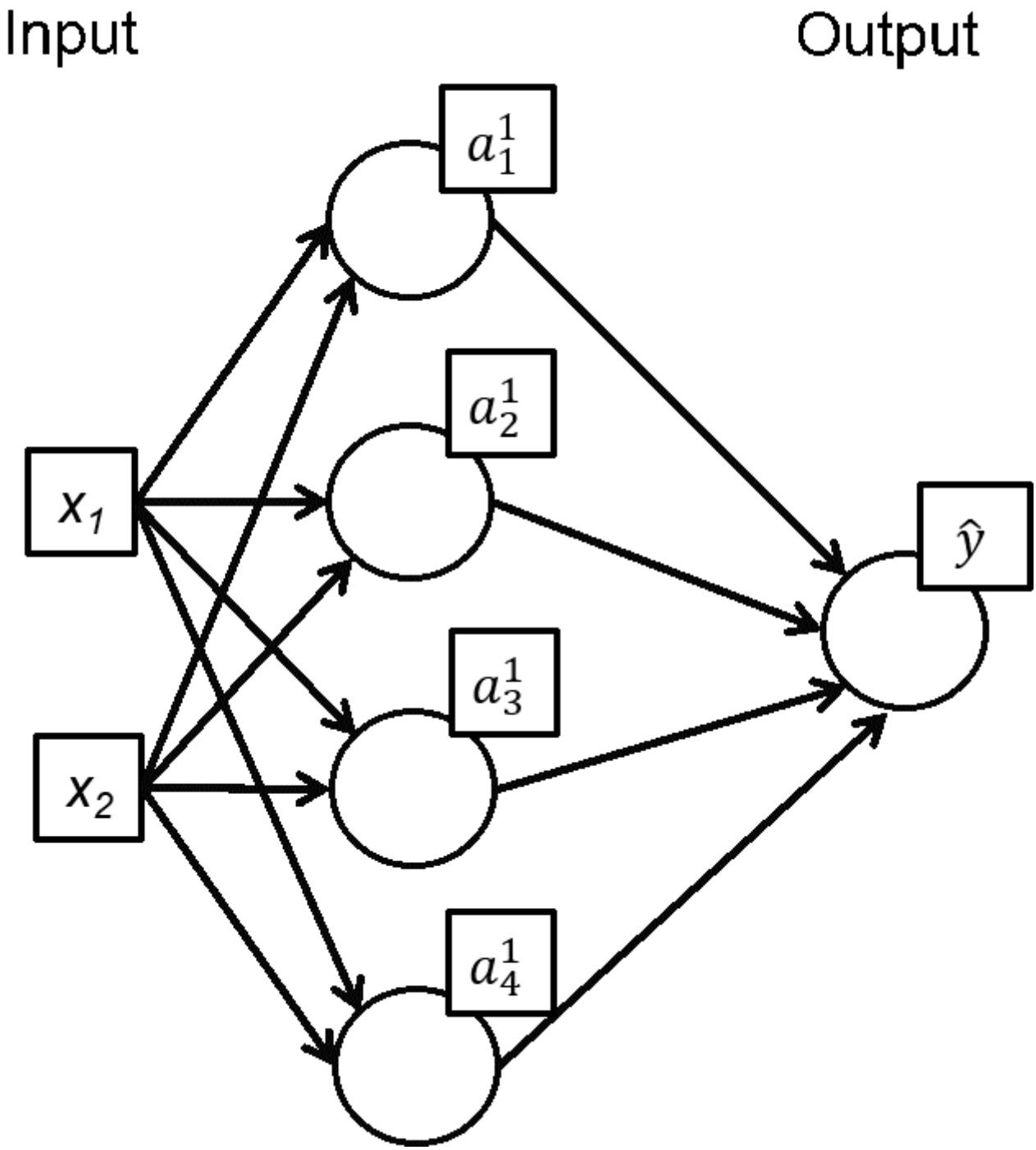
b. Use case: Regression, but Non-Negative.

## vi. Softmax:

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 $\eta$ 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:

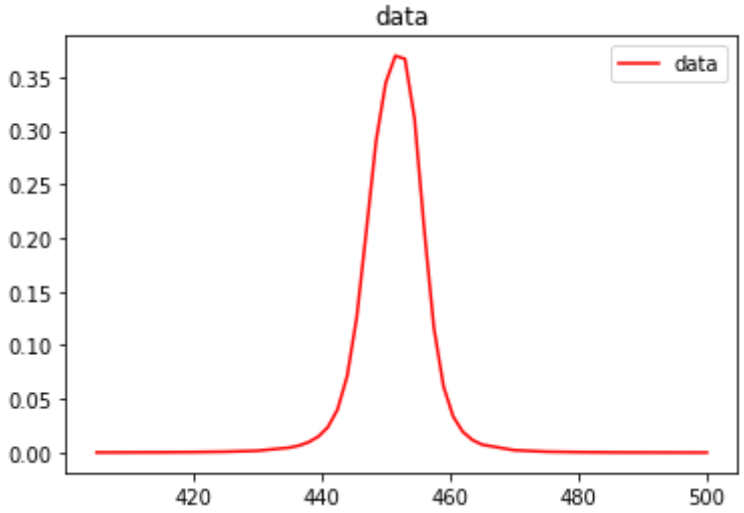
| **Input** $x = (x_1, x_2)^T = a^0$ | **PSP** $z^1$ | **Activation** $a^1$ | **Ouput** $\hat{y} = a^2$ |
|---|---|---|---|
| $(-1, 1)^T$ | | | |
| $(2, -1)^T$ | | | |

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** $b^2(t=1)$ |
|---|---|---|---|

# Answer

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, RMSpro
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_sc


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

num_inputs = X_train.shape[1] # should be 1 in case of Eckerle4
num_hidden = [20,11,7] # for each hidden layer: number of hidden units in form of a
num_outputs = 1 # predict single number in case of Eckerle4

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

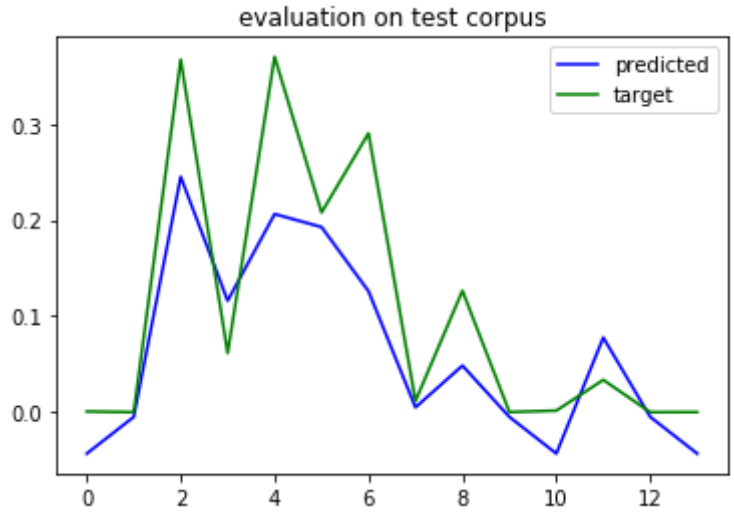
```
# Sequential network structure.
model = Sequential()

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))

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 = SGD(learning_rate=0.1)
model.compile(loss='mse', optimizer=opt, metrics=["mean_squared_error"])# FIX!!!

# Training model with train data. Fixed random seed:
np.random.seed(3)
num_epochs = 215# FIX !!!
batch_size = 3 # FIX !!!
history = model.fit(X_train, y_train, epochs=num_epochs, batch_size=batch_size, ver

###-----------
# 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])
```

# Answer

## Configuration 1

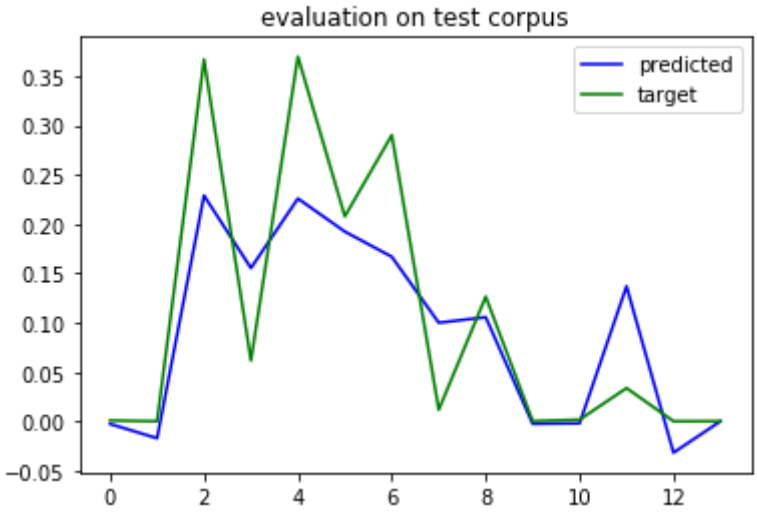
num_hidden = [20,11,7] dropout = 0 num_epochs = 215 batch_size = 3 Activation Function = relu learning rate = 0.1 loss function = mean squared error optimizer = SGD batch size = 3




Mean = 0.0587084 Standard deviation = 0.0101029

## Configuration 2

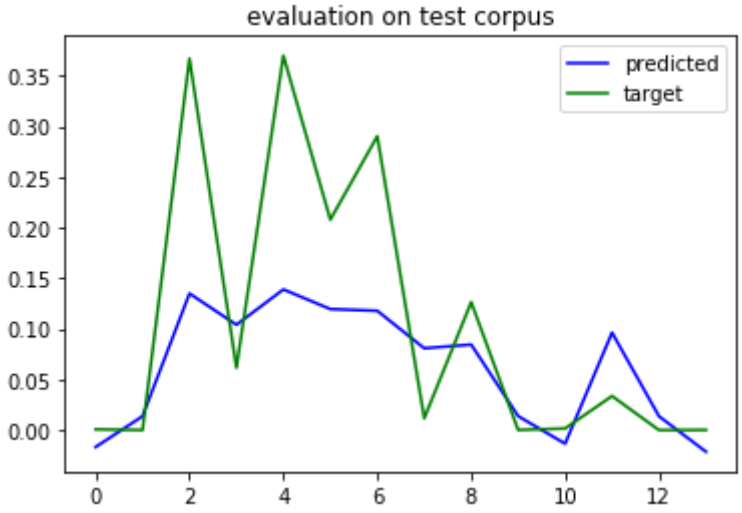
num_hidden = [20,11,7] dropout = 0 num_epochs = 215 batch_size = 3 Activation Function = relu learning rate = 0.1 loss function = mean squared error optimizer = SGD batch size = 5




Mean = 0.0748863 Standard deviation = 0.0126227

## Configuration 3

num_hidden = [25,20,7] dropout = 0 num_epochs = 215 batch_size = 3 Activation Function = relu learning rate = 0.1 loss function = mean squared error optimizer = SGD batch size = 5




Mean = 0.0748863 Standard deviation = 0.0126227

# 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, BatchNormalization
from tensorflow.keras import Model, Input, Sequential
from tensorflow.keras.optimizers import SGD, Adam, Adadelta, Adagrad, Nadam, RMSpro
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_dat

# 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.shap
print("validation input shape: %s, validation target shape: %s"  % (validation_inpu
print("test input shape: %s, test target shape: %s"  % (test_input.shape, test_targ
# 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.sha
validation_input = validation_input.reshape(validation_input.shape[0], validation_i
test_input = test_input.reshape(test_input.shape[0], test_input.shape[1] * test_inp
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': [0.25], # numpy array, e.g. [0.1, 0.2]
         'hiddenLayerSizes': [250], # as before
         'solver': 'Adam',
         'activation':'relu'},
     {'learningRates': [0.15], # numpy array, e.g. [0.1, 0.2]
         'hiddenLayerSizes': [200], # as before
         'solver': 'Nadam',
         'activation':'relu'},
    {'learningRates': [0.15,0.17], # numpy array, e.g. [0.1, 0.2]
         'hiddenLayerSizes': [100], # as before
         'solver': 'RMSprop',
         'activation':'relu'},
    {'learningRates': [0.1,0.2], # numpy array, e.g. [0.1, 0.2]
         'hiddenLayerSizes': [55,60], # as before
         'solver': 'SGD',
         'activation':'relu'},
```

```
    {'learningRates': [0.35,0.37], # numpy array, e.g. [0.1, 0.2]
        'hiddenLayerSizes': [50,30], # as before
        'solver': 'Adadelta',
        'activation':'relu'},
    {'learningRates': [0.37,0.38], # numpy array, e.g. [0.1, 0.2]
        'hiddenLayerSizes': [50,30], # as before
        'solver': 'Adagrad',
        'activation':'relu'},

   # activation of hidden layers


]

learningRateSchedule = False # FIX!!! True: apply (exponential) learning rate sched
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 apply
regularizer = tfr.l1(l=regularization_weight) # or l2 or l1_l2; used for both weigh
momentum = 0.9 # FIX!!! 0 or e.g. 0.9, 0.99; ONLY FOR STOCHASTIC GRADIENT DESCENT A
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, 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_regularizer=regula
  #  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=regularize

  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 = learningRate
      print("... applying exponential decay learning rate schedule with initial lea
    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 lea
      elif solver == 'Adadelta':
        opt = Adadelta(learning_rate=lr_schedule)
```

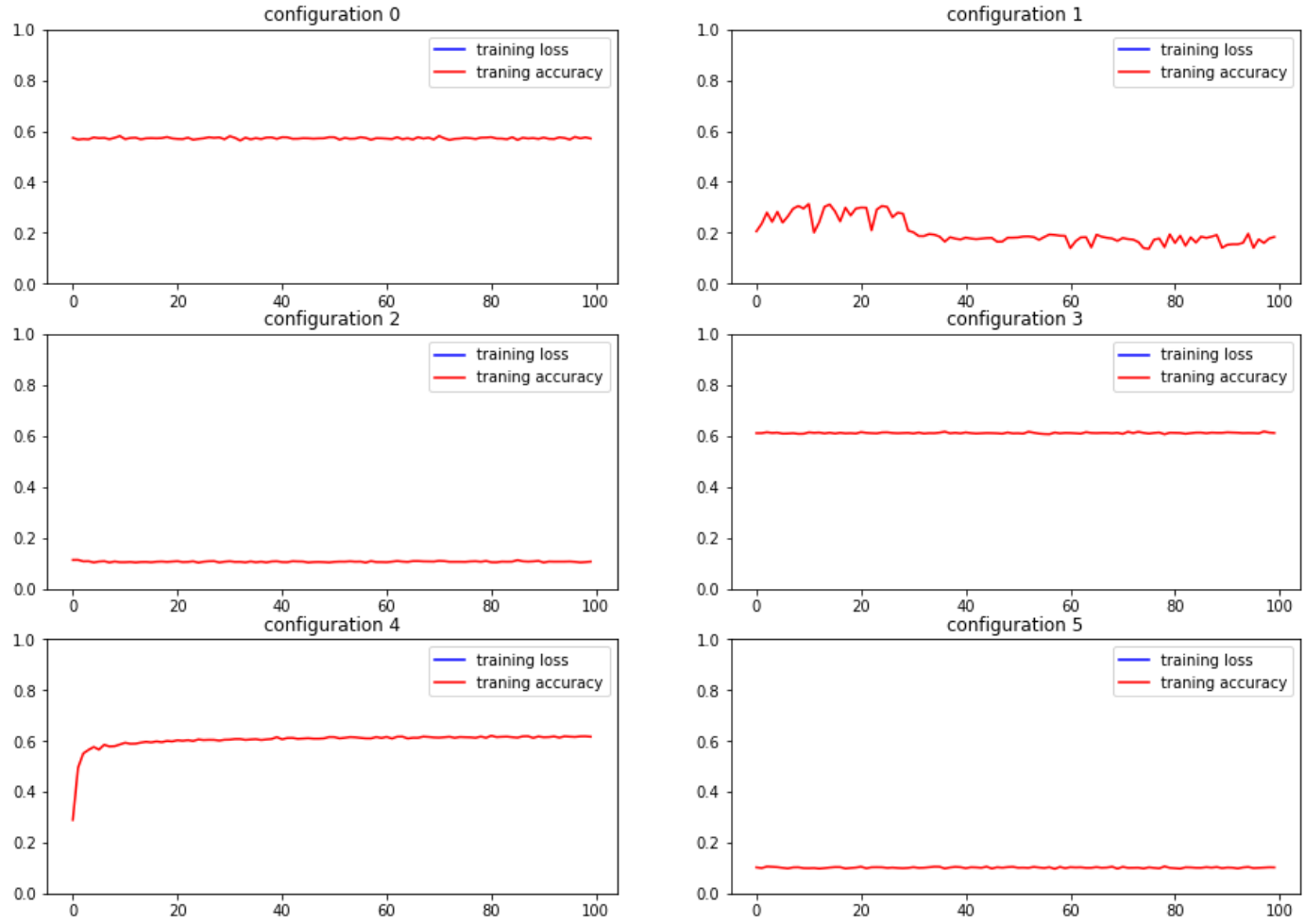
```
        elif solver == 'Adagrad':
          opt = Adagrad(learning_rate=lr_schedule)
        elif solver == 'RMSprop':
          opt = RMSprop(learning_rate=lr_schedule, momentum = momentum)
        model.compile(optimizer=opt,loss=tf.keras.losses.SparseCategoricalCrossentrop

        # 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, batch
        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_epoch
        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'], co
                 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 repet
    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_r

  # 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, hspace=
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'], c
              label = 'traning accuracy')
  axes[axes_indices[i]].legend()

  i = i + 1


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

# Answer

## FINDINGS OF Experiment 1:



configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [50, 30], 'solver': 'SGD', 'activation': 'relu'}:

optimal learning rate: 0.100000

final training loss: 2.422811

final training accuracy: 0.570800

final validation loss: 1.858070

final validation accuracy: 0.817100

final test loss: 1.862645

final test accuracy: 0.805600

configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [50, 30], 'solver': 'Adam', 'activation': 'relu'}:

optimal learning rate: 0.100000

final training loss: 6.640169

final training accuracy: 0.183700

final validation loss: 6.615485

final validation accuracy: 0.234500

final test loss: 6.612063

final test accuracy: 0.237500

configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [50, 30], 'solver': 'Nadam', 'activation': 'relu'}:

optimal learning rate: 0.100000

final training loss: 9.630845

final training accuracy: 0.105800

final validation loss: 6.954296

final validation accuracy: 0.109000

final test loss: 6.954338

final test accuracy: 0.102800

configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [50, 30], 'solver': 'Adadelta', 'activation': 'relu'}:

optimal learning rate: 0.120000

final training loss: 1.973515

final training accuracy: 0.611920

final validation loss: 1.622954

final validation accuracy: 0.837400

final test loss: 1.616173

final test accuracy: 0.835000

configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [50, 30], 'solver': 'Adagrad', 'activation': 'relu'}:

optimal learning rate: 0.110000

final training loss: 2.136097

final training accuracy: 0.616440

final validation loss: 1.633894

final validation accuracy: 0.831900

final test loss: 1.631751

final test accuracy: 0.824400

configuration {'learningRates': [0.1, 0.11, 0.12], 'hiddenLayerSizes': [50, 30], 'solver': 'RMSprop', 'activation': 'relu'}:

optimal learning rate: 0.110000

final training loss: 129.548080

final training accuracy: 0.102080

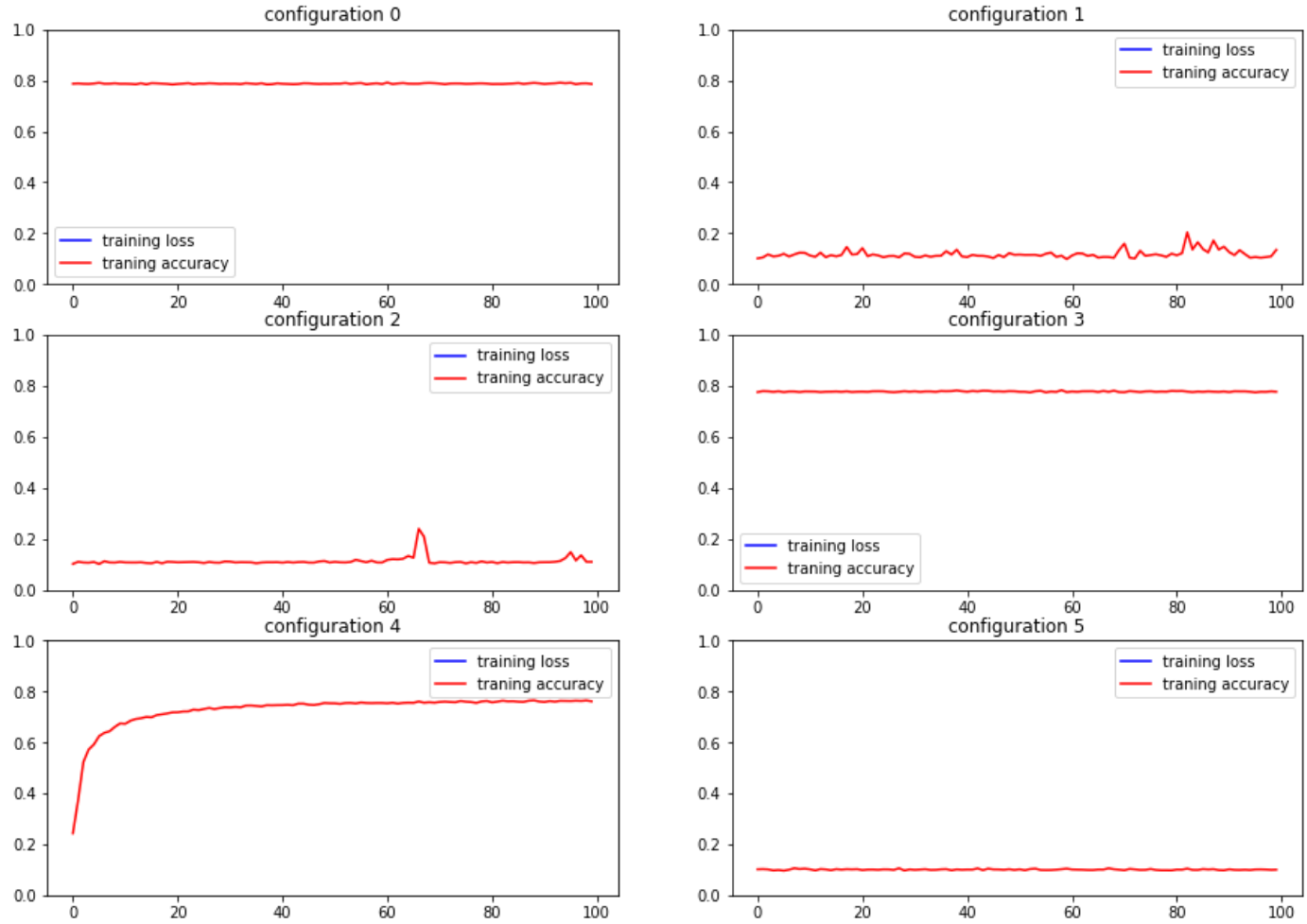
final validation loss: 83.609398

final validation accuracy: 0.176700

final test loss: 83.630838

final test accuracy: 0.161900

## FINDINGS OF EXPERIMENT 2:




configuration {'learningRates': [0.1, 0.2], 'hiddenLayerSizes': [55, 60], 'solver': 'SGD', 'activation': 'relu'}:

optimal learning rate: 0.100000

final training loss: 1.068864

final training accuracy: 0.786400

final validation loss: 0.772645

final validation accuracy: 0.929800

final test loss: 0.778956

final test accuracy: 0.925200

configuration {'learningRates': [0.2, 0.3], 'hiddenLayerSizes': [55, 60], 'solver': 'Adam', 'activation': 'relu'}:

optimal learning rate: 0.200000

final training loss: 11.740863

final training accuracy: 0.134800

final validation loss: 10.985705

final validation accuracy: 0.140900

final test loss: 10.985461

final test accuracy: 0.150000

configuration {'learningRates': [0.3, 0.31], 'hiddenLayerSizes': [50, 30], 'solver': 'Nadam', 'activation': 'relu'}:

optimal learning rate: 0.300000

final training loss: 56.724408

final training accuracy: 0.109160

final validation loss: 33.763234

final validation accuracy: 0.106400

final test loss: 33.761829

final test accuracy: 0.113500

configuration {'learningRates': [0.35, 0.37], 'hiddenLayerSizes': [50, 30], 'solver': 'Adadelta', 'activation': 'relu'}:

optimal learning rate: 0.370000

final training loss: 1.096682

final training accuracy: 0.777400

final validation loss: 0.782862

final validation accuracy: 0.931700

final test loss: 0.788440

final test accuracy: 0.926000

configuration {'learningRates': [0.37, 0.38], 'hiddenLayerSizes': [50, 30], 'solver': 'Adagrad', 'activation': 'relu'}:

optimal learning rate: 0.370000

final training loss: 1.146591

final training accuracy: 0.761480

final validation loss: 0.831216

final validation accuracy: 0.914600

final test loss: 0.838594

final test accuracy: 0.909600

configuration {'learningRates': [0.39, 0.4], 'hiddenLayerSizes': [50, 30], 'solver': 'RMSprop', 'activation': 'relu'}:

optimal learning rate: 0.400000

final training loss: 1570.586729

final training accuracy: 0.100360

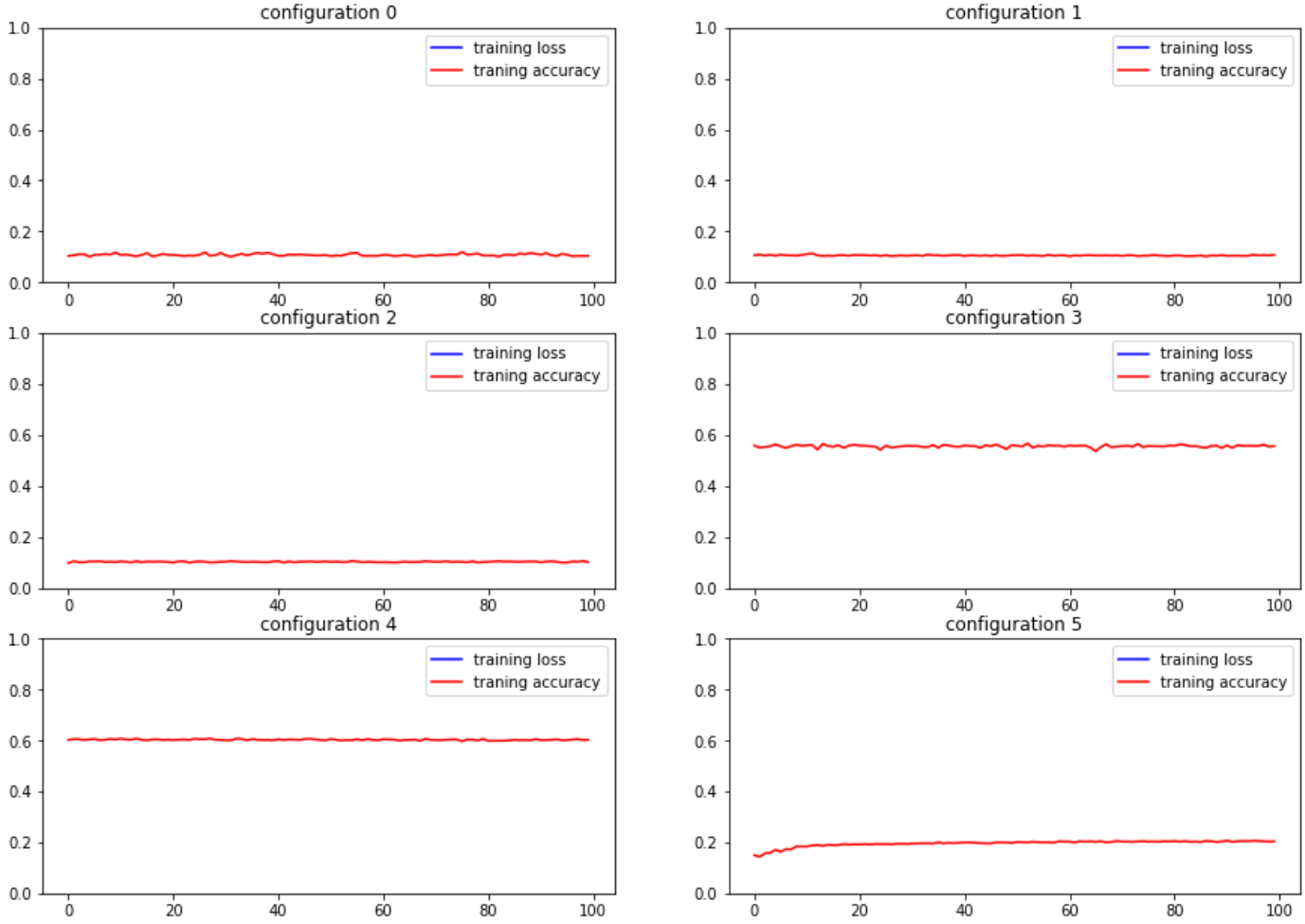
final validation loss: 1333.120870

final validation accuracy: 0.129100

final test loss: 1347.978428

final test accuracy: 0.112300

## FINDINGS OF EXPERIMENT 3:




configuration {'learningRates': [0.25], 'hiddenLayerSizes': [300, 200, 30], 'solver': 'Adam', 'activation': 'relu'}:

optimal learning rate: 0.250000

final training loss: 107.403180

final training accuracy: 0.103360

final validation loss: 106.208815

final validation accuracy: 0.125800

final test loss: 106.209188

final test accuracy: 0.119200

configuration {'learningRates': [0.15], 'hiddenLayerSizes': [300, 200, 30], 'solver': 'Nadam', 'activation': 'relu'}:

optimal learning rate: 0.150000

final training loss: 81.041597

final training accuracy: 0.107560

final validation loss: 64.303141

final validation accuracy: 0.109000

final test loss: 64.303326

final test accuracy: 0.102800

configuration {'learningRates': [0.15, 0.17], 'hiddenLayerSizes': [100, 70, 30], 'solver': 'RMSprop', 'activation': 'relu'}:

optimal learning rate: 0.150000

final training loss: 210.109740

final training accuracy: 0.100480

final validation loss: 187.408234

final validation accuracy: 0.099100

final test loss: 187.408264

final test accuracy: 0.098000

configuration {'learningRates': [0.1, 0.2], 'hiddenLayerSizes': [55, 60], 'solver': 'SGD', 'activation': 'relu'}:

optimal learning rate: 0.100000

final training loss: 2.494765

final training accuracy: 0.556440

final validation loss: 1.925437

final validation accuracy: 0.774300

final test loss: 1.928001

final test accuracy: 0.763800

configuration {'learningRates': [0.35, 0.37], 'hiddenLayerSizes': [50, 30], 'solver': 'Adadelta', 'activation': 'relu'}:

optimal learning rate: 0.370000

final training loss: 2.205468

final training accuracy: 0.602980

final validation loss: 1.838857

final validation accuracy: 0.825900

final test loss: 1.843558

final test accuracy: 0.811300

configuration {'learningRates': [0.37, 0.38], 'hiddenLayerSizes': [50, 30], 'solver': 'Adagrad', 'activation': 'relu'}:

optimal learning rate: 0.370000

final training loss: 3.172650

final training accuracy: 0.203840

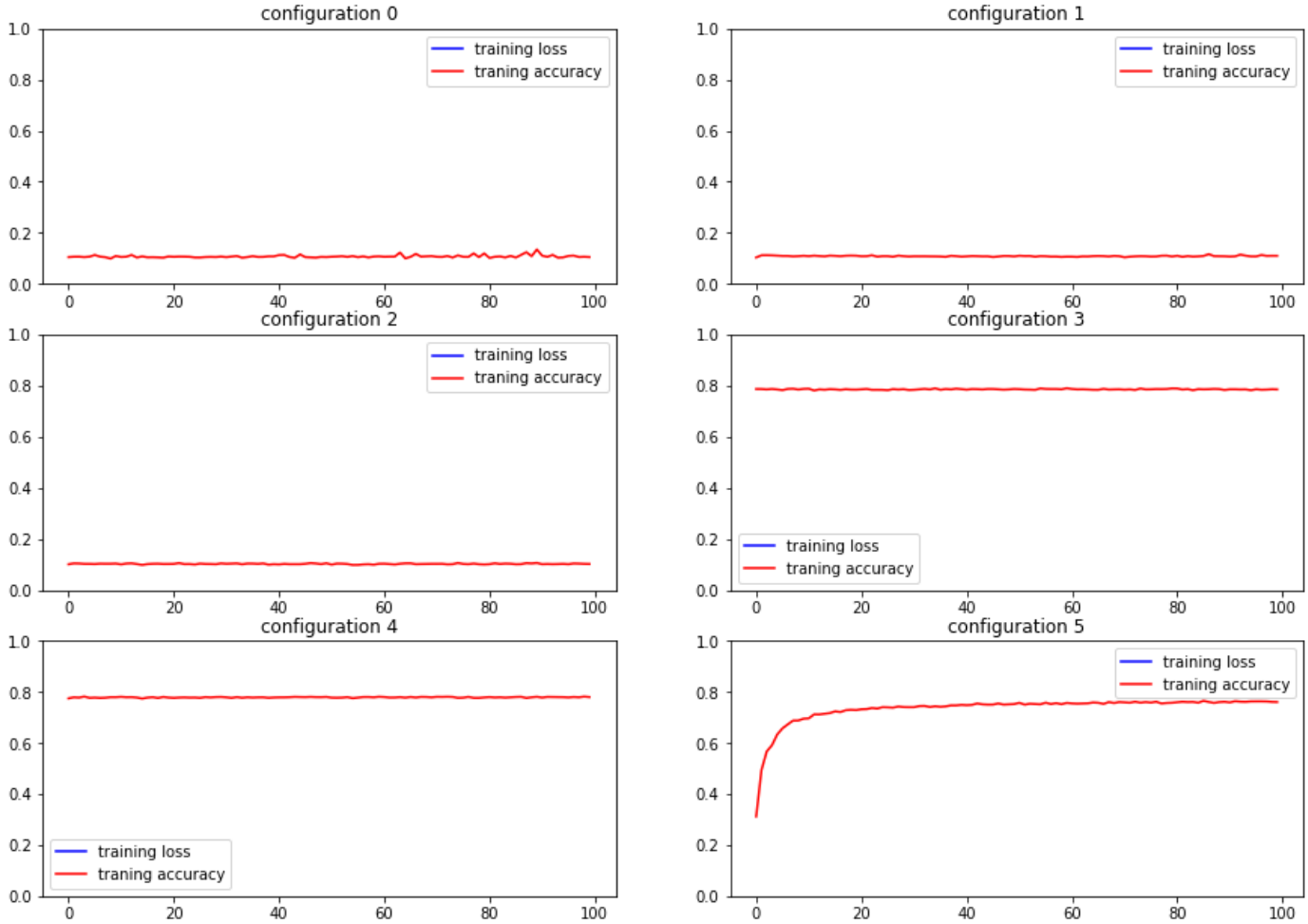
final validation loss: 3.133374

final validation accuracy: 0.259400

final test loss: 3.127857

final test accuracy: 0.264900

## FINDINGS OF EXPERIMENT 4:




configuration {'learningRates': [0.25], 'hiddenLayerSizes': [300, 200, 30], 'solver': 'Adam', 'activation': 'relu'}:

optimal learning rate: 0.250000

final training loss: 54.234446

final training accuracy: 0.104540

final validation loss: 31.805575

final validation accuracy: 0.181600

final test loss: 31.804397

final test accuracy: 0.186000

configuration {'learningRates': [0.15], 'hiddenLayerSizes': [300, 200, 30], 'solver': 'Nadam', 'activation': 'relu'}:

optimal learning rate: 0.150000

final training loss: 102.505118

final training accuracy: 0.109700

final validation loss: 74.141925

final validation accuracy: 0.106400

final test loss: 74.141266

final test accuracy: 0.113500

configuration {'learningRates': [0.15, 0.17], 'hiddenLayerSizes': [100, 70, 30], 'solver': 'RMSprop', 'activation': 'relu'}:

optimal learning rate: 0.150000

final training loss: 122.977712

final training accuracy: 0.101700

final validation loss: 144.287500

final validation accuracy: 0.109000

final test loss: 144.290932

final test accuracy: 0.102800

configuration {'learningRates': [0.1, 0.2], 'hiddenLayerSizes': [55, 60], 'solver': 'SGD', 'activation': 'relu'}:

optimal learning rate: 0.100000

final training loss: 1.068421

final training accuracy: 0.785920

final validation loss: 0.773372

final validation accuracy: 0.931600

final test loss: 0.781418

final test accuracy: 0.926200

configuration {'learningRates': [0.35, 0.37], 'hiddenLayerSizes': [50, 30], 'solver': 'Adadelta', 'activation': 'relu'}:

optimal learning rate: 0.350000

final training loss: 1.101753

final training accuracy: 0.780000

final validation loss: 0.786466

final validation accuracy: 0.931700

final test loss: 0.791812

final test accuracy: 0.926200

configuration {'learningRates': [0.37, 0.38], 'hiddenLayerSizes': [50, 30], 'solver': 'Adagrad', 'activation': 'relu'}:

optimal learning rate: 0.370000

final training loss: 1.154554

final training accuracy: 0.760640

final validation loss: 0.817311

final validation accuracy: 0.921500

final test loss: 0.823596

final test accuracy: 0.917200

# 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 [ ]:

```
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, BatchNormalization
from tensorflow.keras import Model, Input, Sequential
from tensorflow.keras.optimizers import SGD, Adam, Adadelta, Adagrad, Nadam, RMSpro
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_dat

# 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.shap
print("validation input shape: %s, validation target shape: %s"  % (validation_inpu
print("test input shape: %s, test target shape: %s"  % (test_input.shape, test_targ
# 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.sha
validation_input = validation_input.reshape(validation_input.shape[0], validation_i
test_input = test_input.reshape(test_input.shape[0], test_input.shape[1] * test_inp
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 = [15,25,50,75,100,150] # 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 = initialLearningR

solver = 'SGD'
activation = 'relu' # 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. glorot_
bias_init = tfi.Zeros() # FIX!!! default: Zeros(); for some possible values see wei

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

num_epochs = 30 # FIX !!!
batch_size = 2 # 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, kernel_

# 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_i
#  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_init,

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) # SGD
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(from

# 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) arra
# 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 left
print("Visualization of the weights between input and some of the hidden neurons of
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(())
```

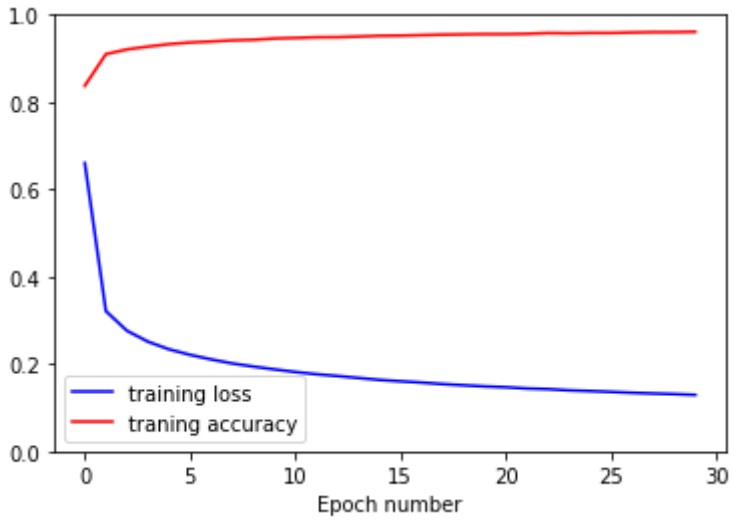
```
    ax.set_yticks(())

plt.show()

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

# 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 = 'tr
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) arra
# 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 left
print("Visualization of the weights between input and some of the hidden neurons of
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()
```

# Answer

## num_hidden = [15] , Epochs = 30 batch_size = 2, AF = sigmoid

number of layers: 2




final training loss: 0.129285

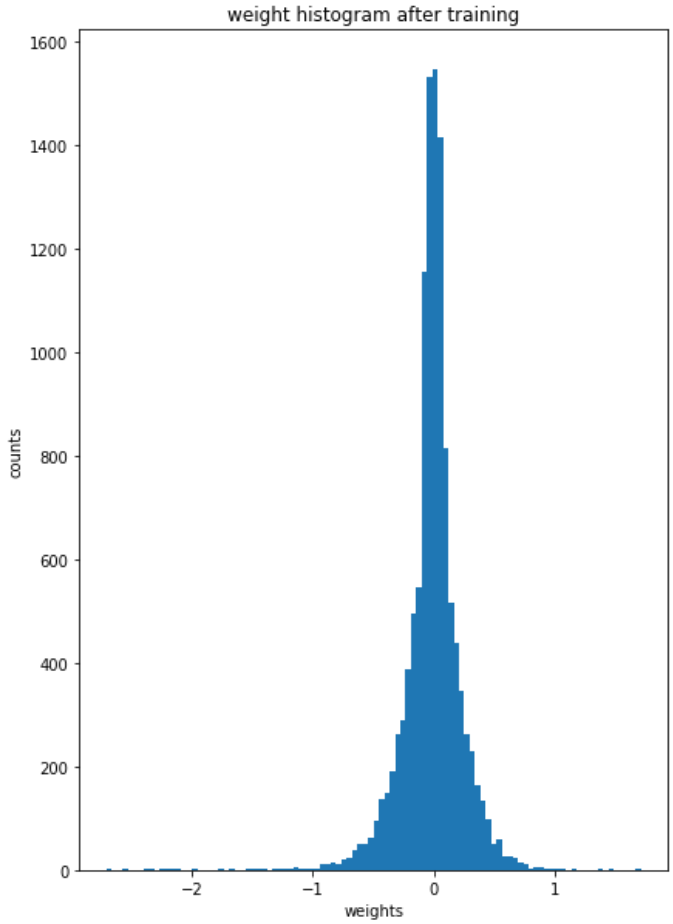
final training accuracy: 0.961020

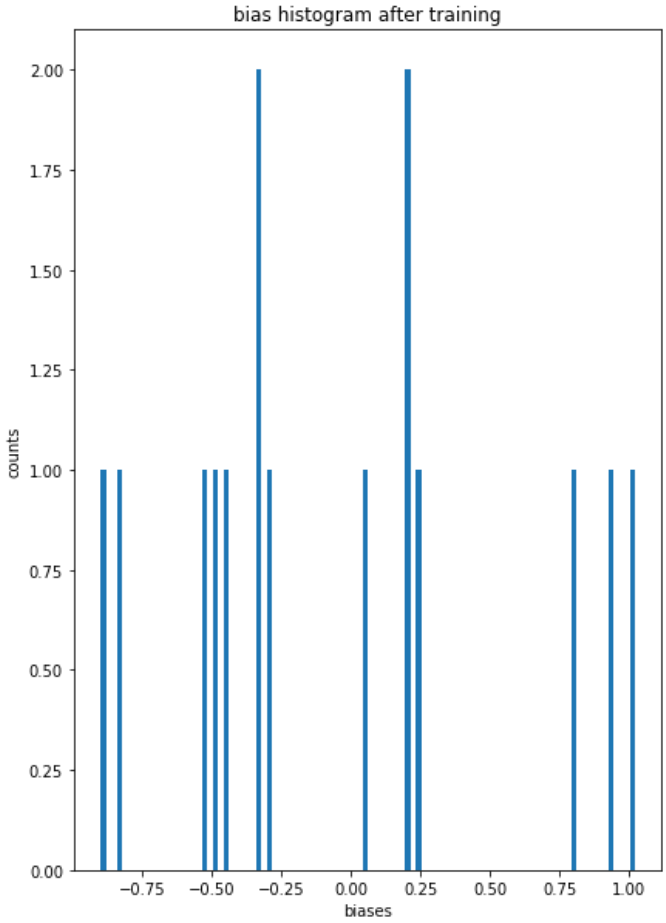
final validation loss: 0.176996

final validation accuracy: 0.949500

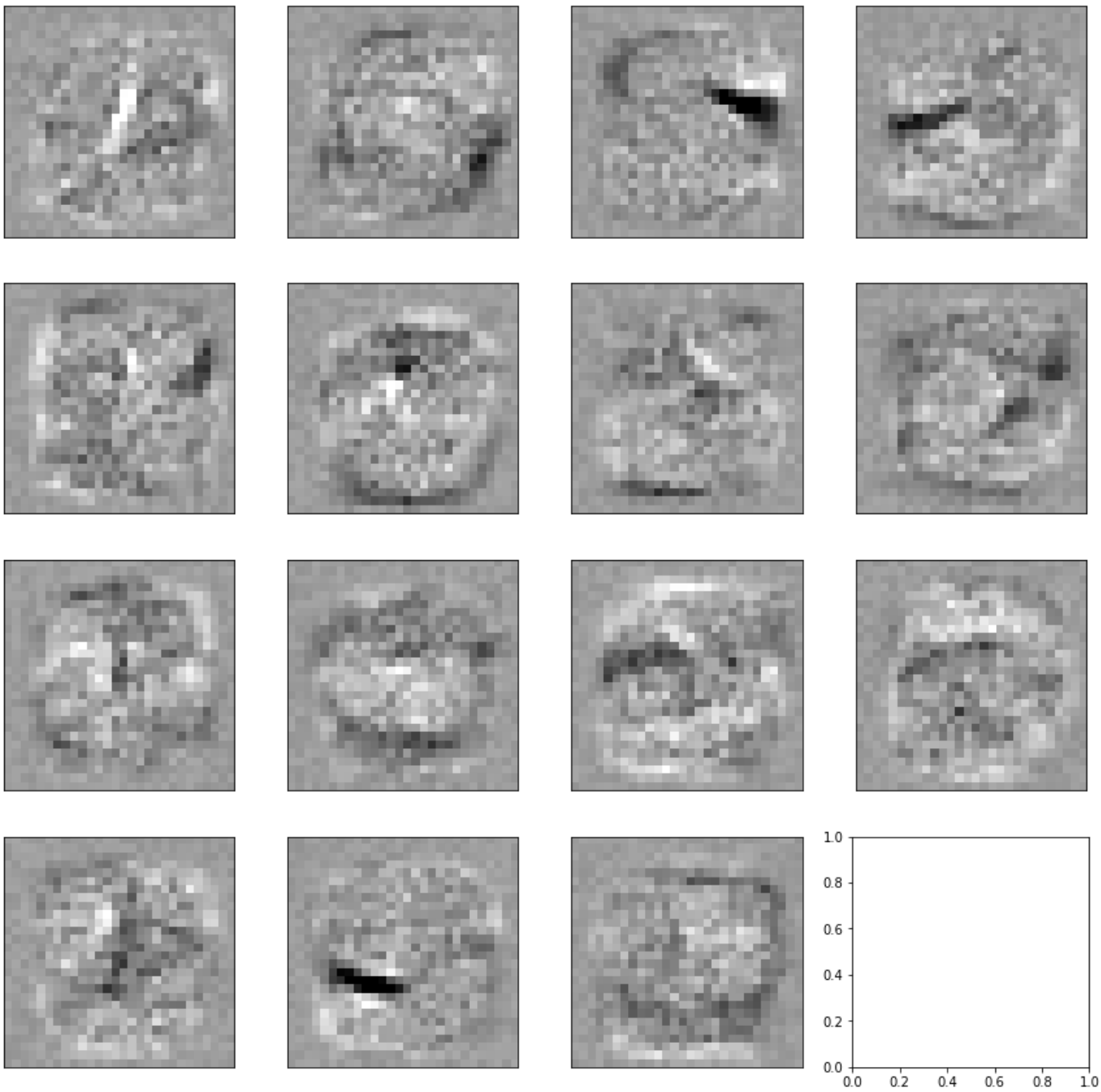
final test loss: 0.168526

final test accuracy: 0.949500



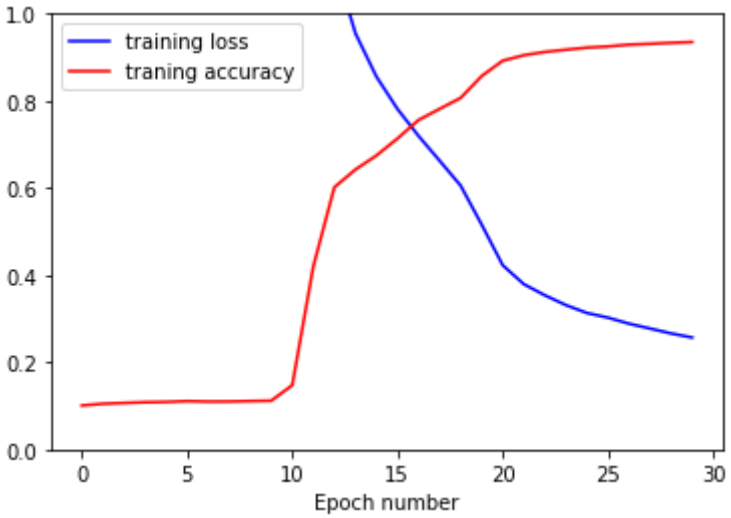




Visualization of the weights between input and some of the hidden neurons of the first hidden layer:




# num_hidden =[15,25,50,75,100], Epochs = 30 batch_size = 2, AF = sigmoid

number of layers: 6




final training loss: 0.256901

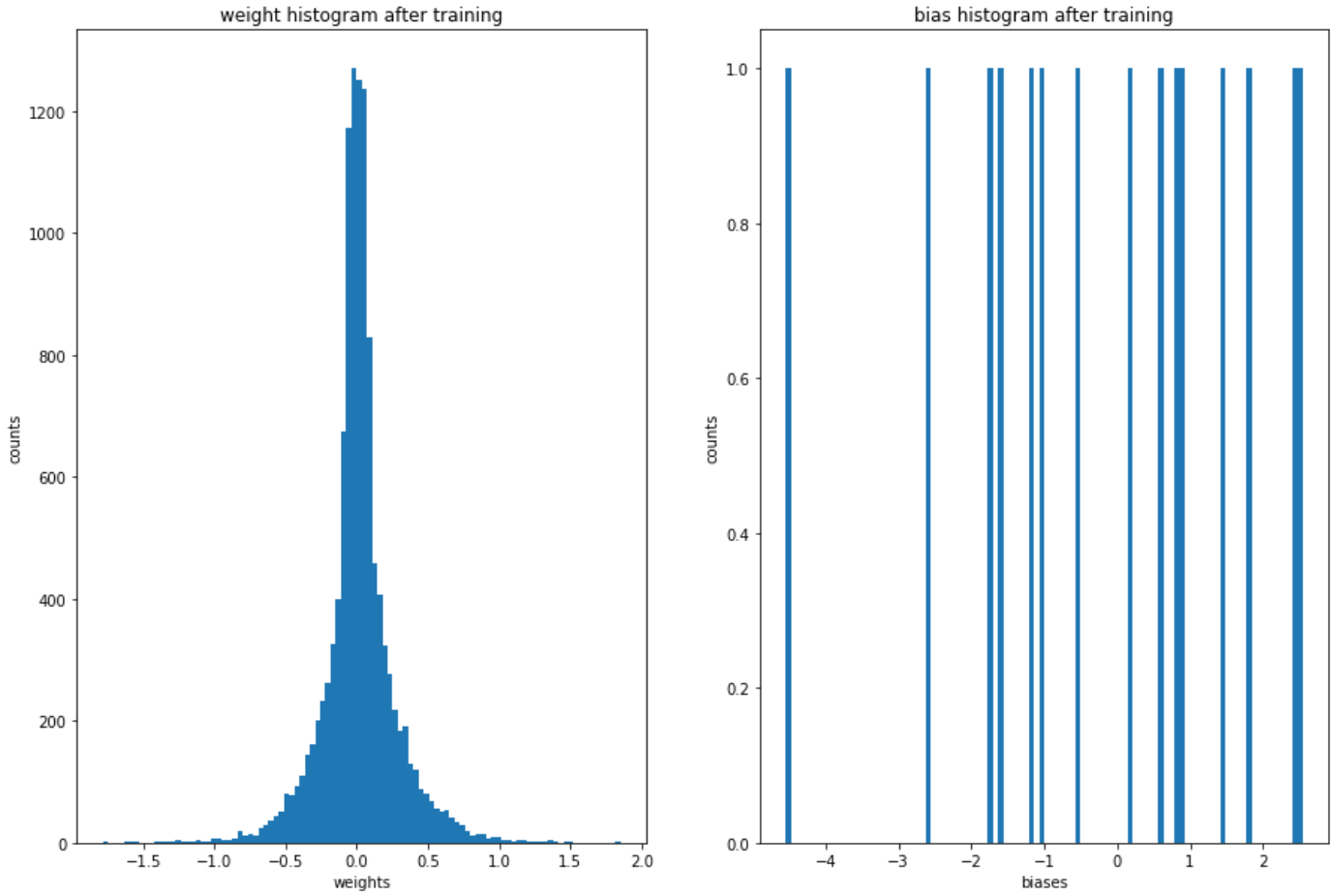
final training accuracy: 0.935100

final validation loss: 0.298294

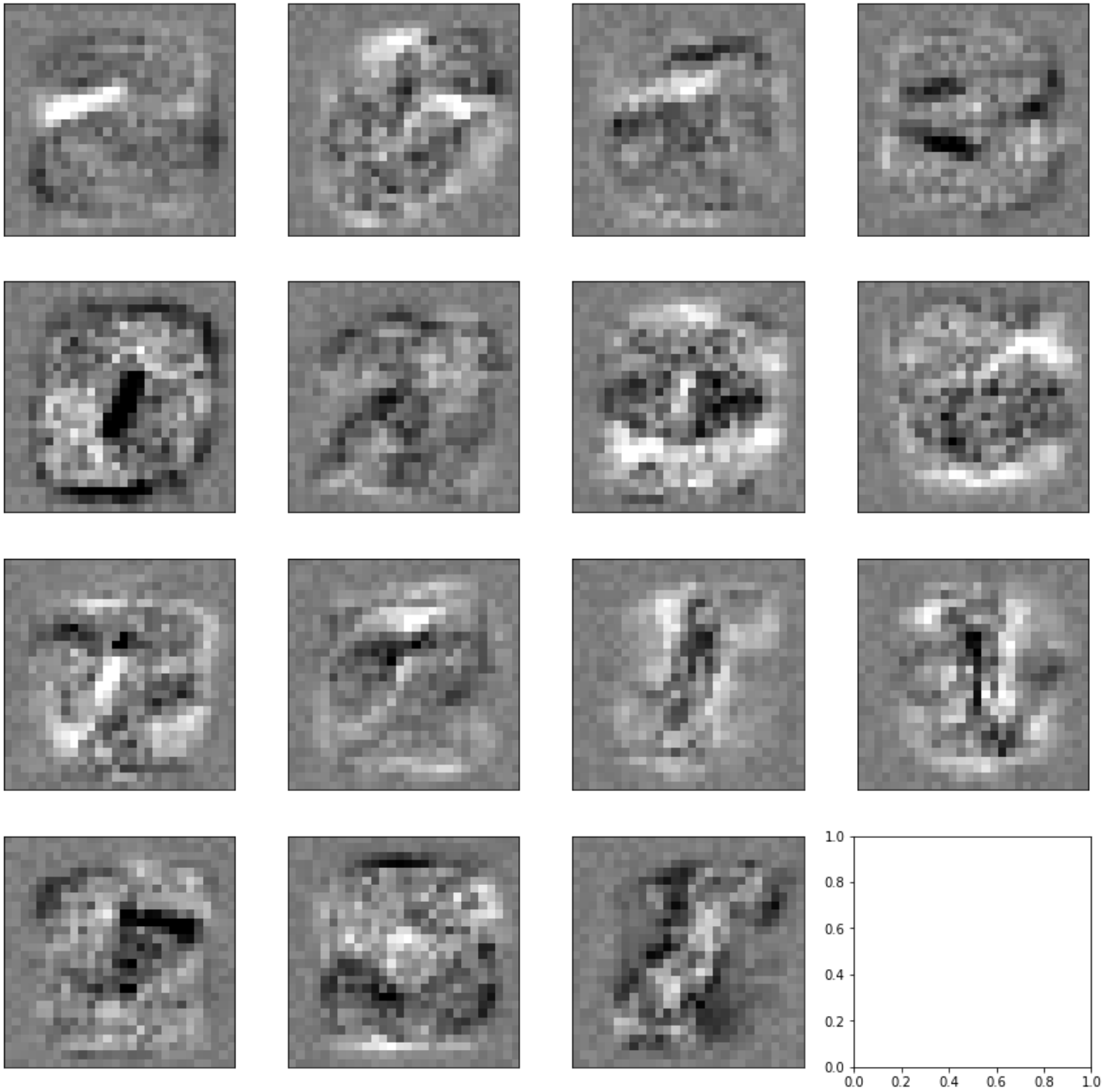
final validation accuracy: 0.928900

final test loss: 0.316086

final test accuracy: 0.924900

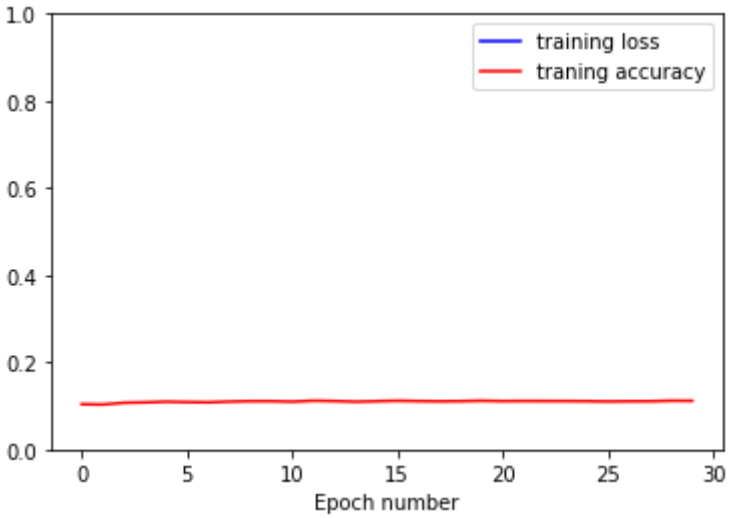



Visualization of the weights between input and some of the hidden neurons of the first hidden layer:




# num_hidden =[15,25,50,75,100,150], Epochs = 30, batch_size = 2, AF = sigmoid

number of layers: 7




final training loss: 2.302258

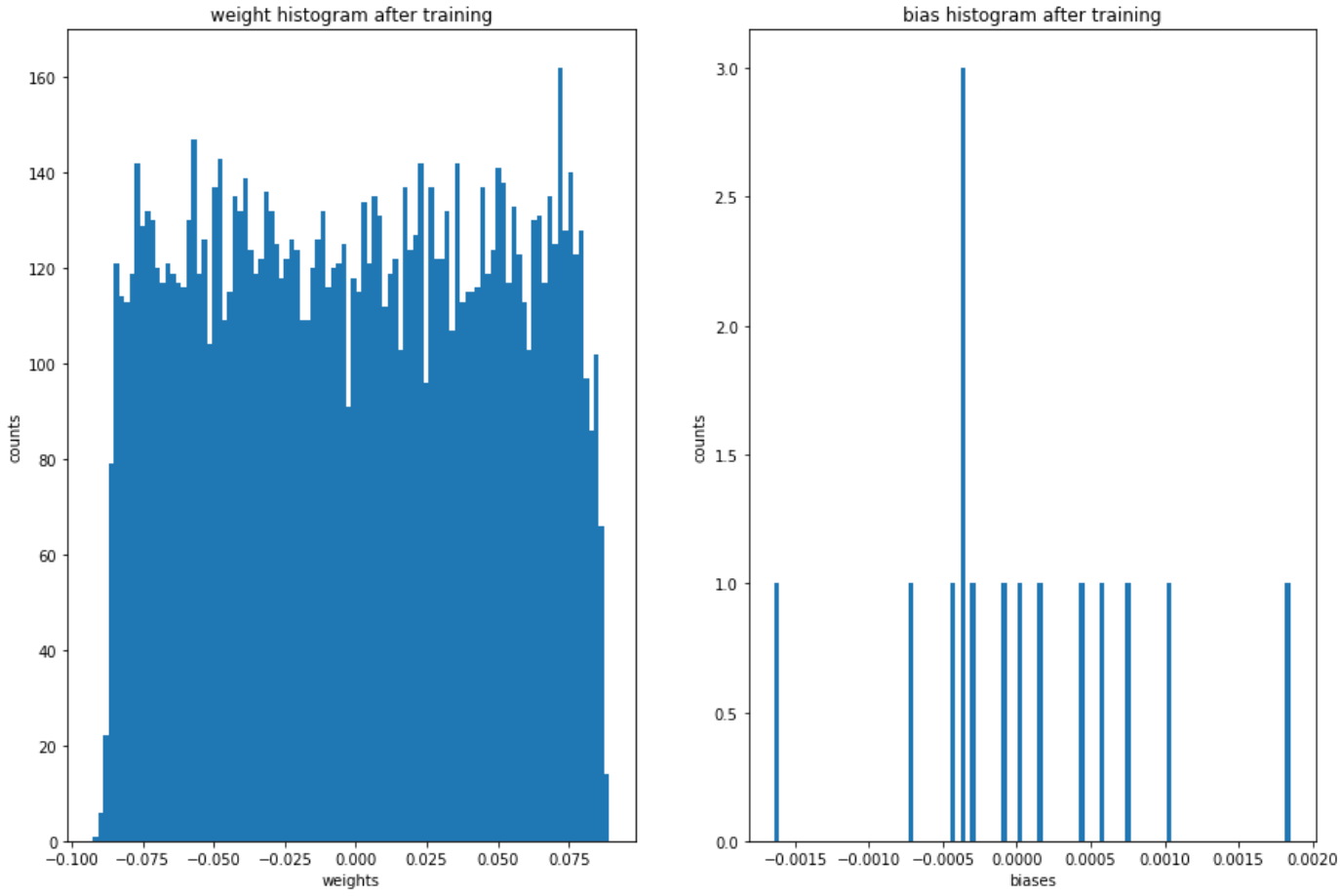
final training accuracy: 0.111620

final validation loss: 2.303587

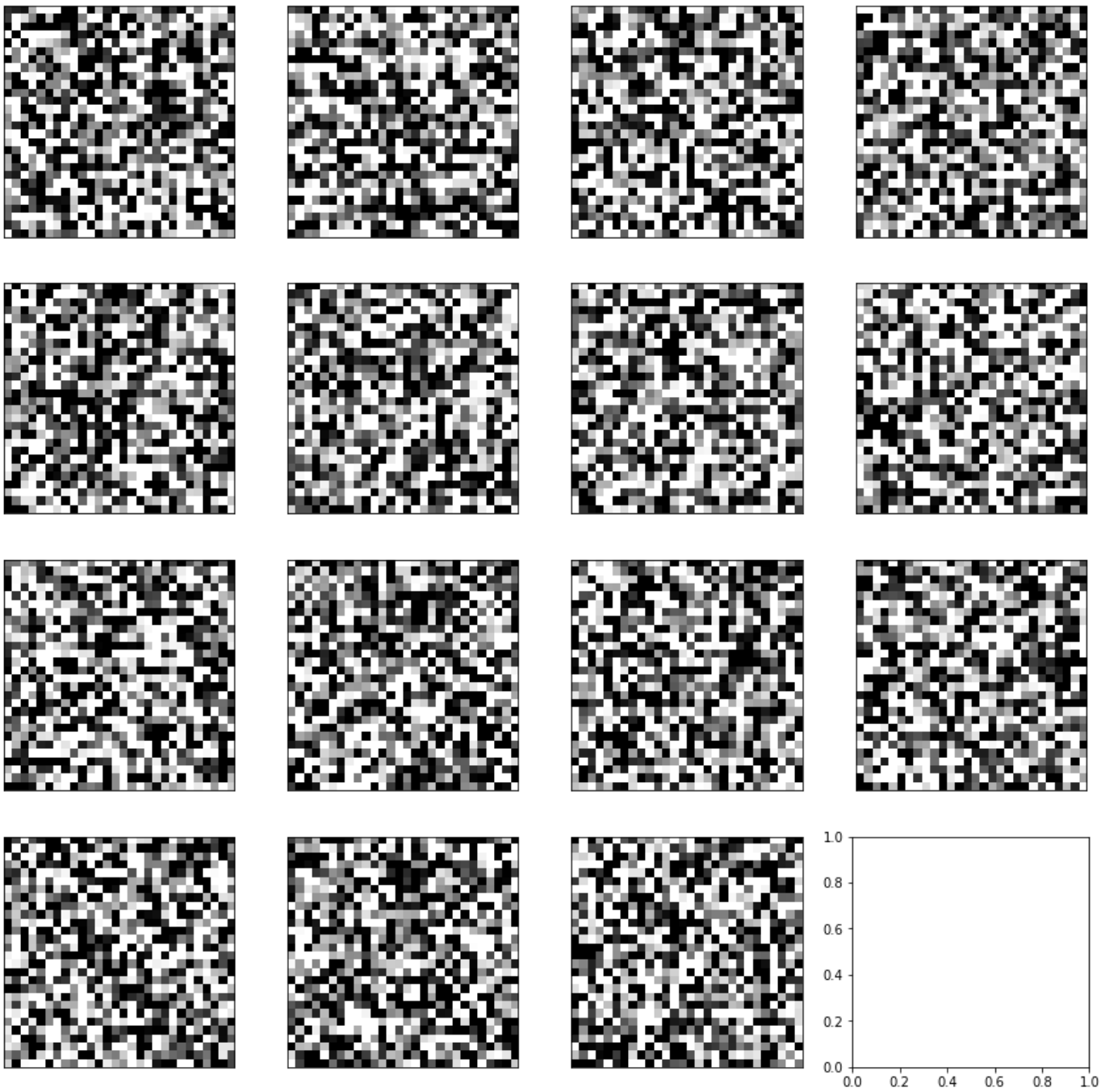
final validation accuracy: 0.106400

final test loss: 2.302365

final test accuracy: 0.113500

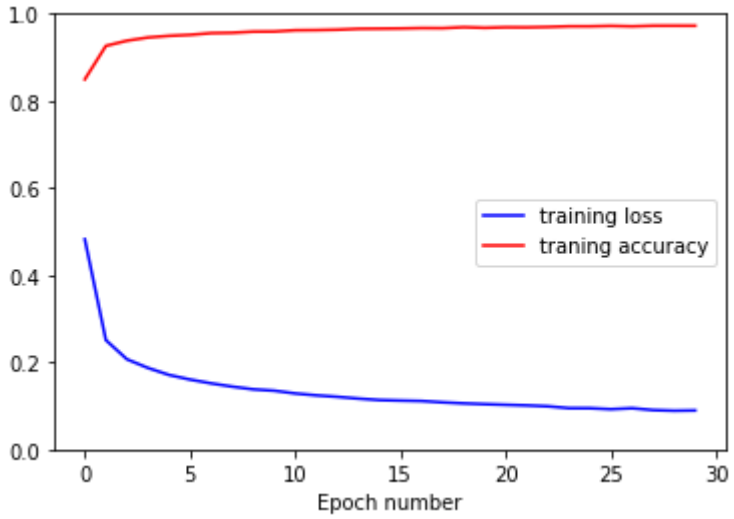



Visualization of the weights between input and some of the hidden neurons of the first hidden layer:




## num_hidden =[15,25,50,75,100,150], Epochs = 30, batch_size = 2, AF = relu

number of layers: 7




final training loss: 0.089496

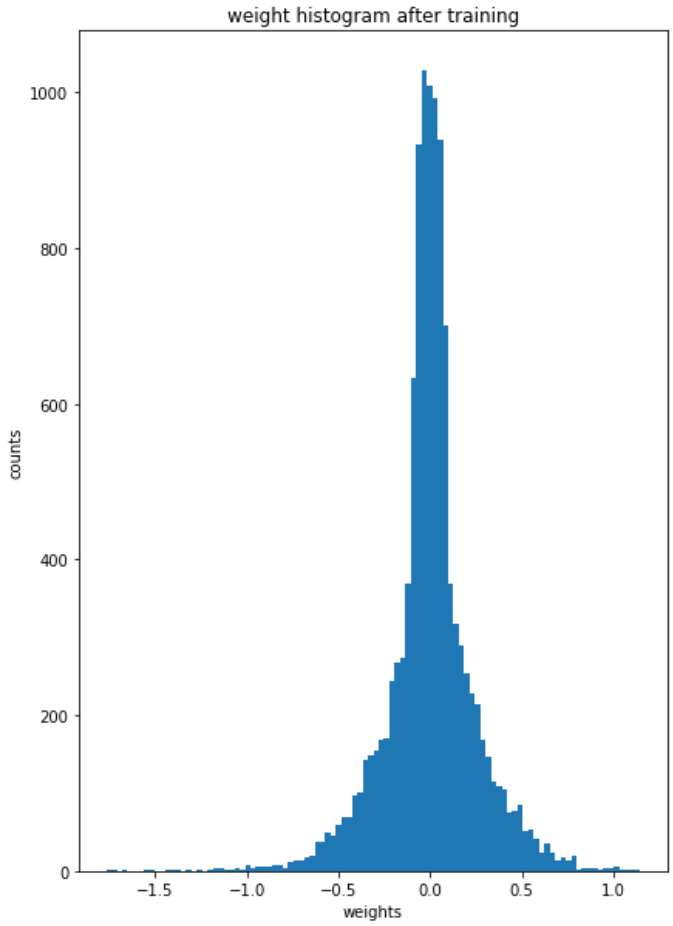
final training accuracy: 0.972440

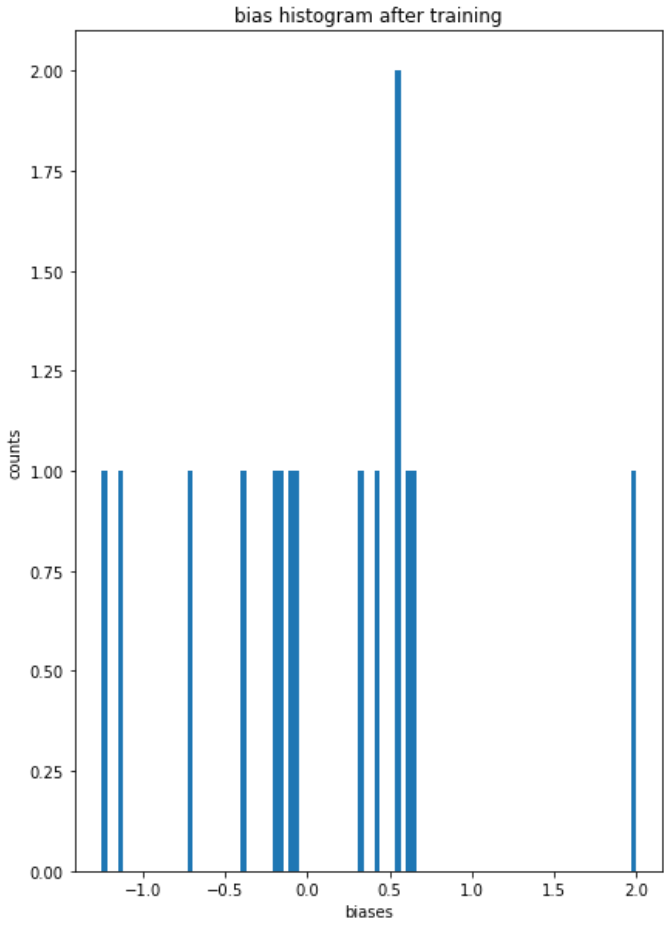
final validation loss: 0.150481

final validation accuracy: 0.963600

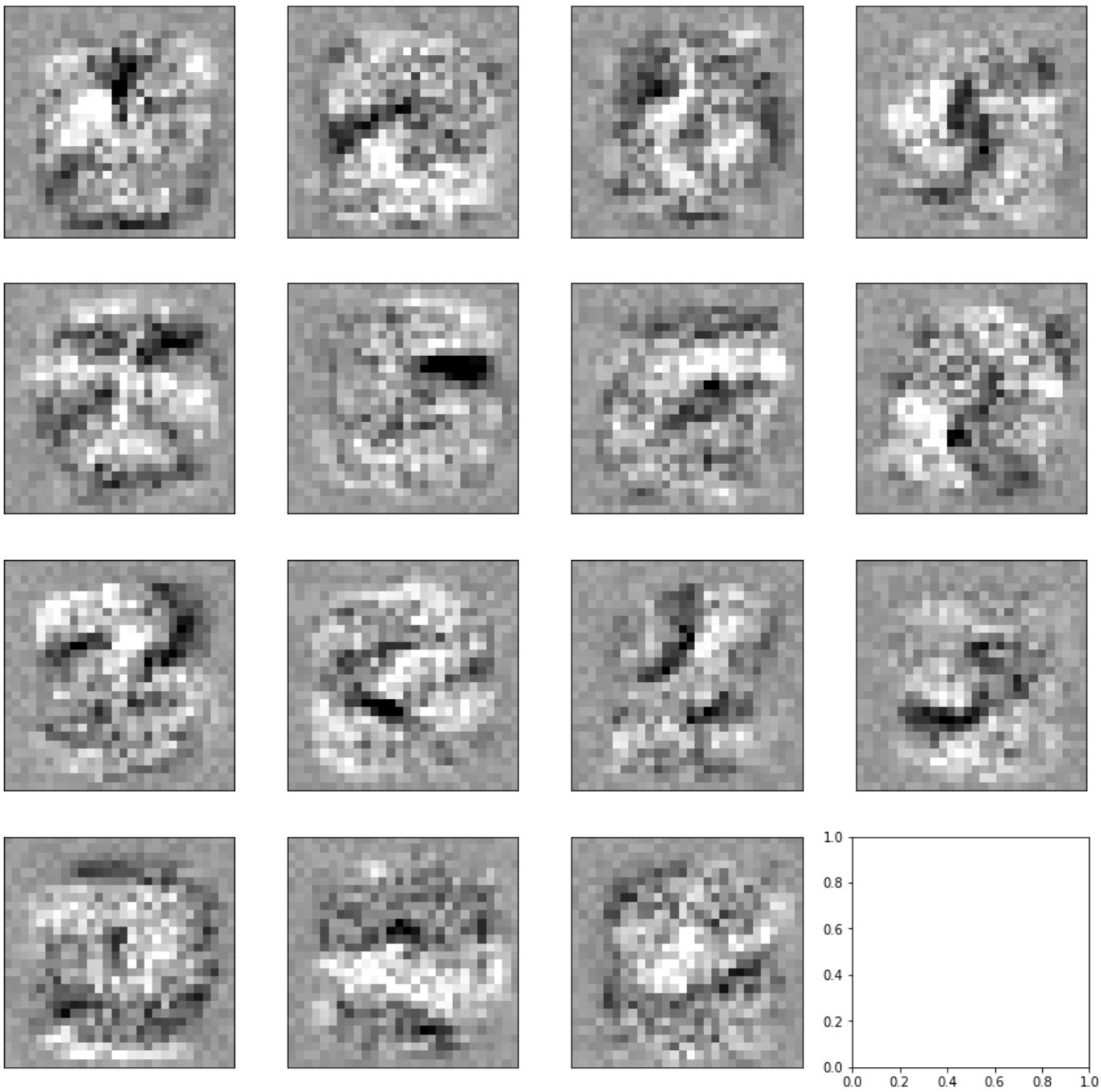
final test loss: 0.170990

final test accuracy: 0.956200




Visualization of the weights between input and some of the hidden neurons of the first hidden layer:




## Findings

| | # hidden layer 1 | # hidden layer 5 | # hidden layer 6 | # hidden layer 6 and ReLU |
|---|---|---|---|---|
| final training loss | 0.129285 | 0.256901 | 2.302258 | 0.089496 |
| final training accuracy | 0.961020 | 0.935100 | 0.111620 | 0.972440 |
| final validation loss | 0.176996 | 0.298294 | 2.303587 | 0.150481 |
| final validation accuracy | 0.949500 | 0.928900 | 0.106400 | 0.963600 |
| final test loss | 0.168526 | 0.316086 | 2.302365 | 0.170990 |
| final test accuracy | 0.949500 | 0.924900 | 0.113500 | 0.956200 |

In short, The network with 6 hidden layer and ReLU activation function is better then others.

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?

# Answer

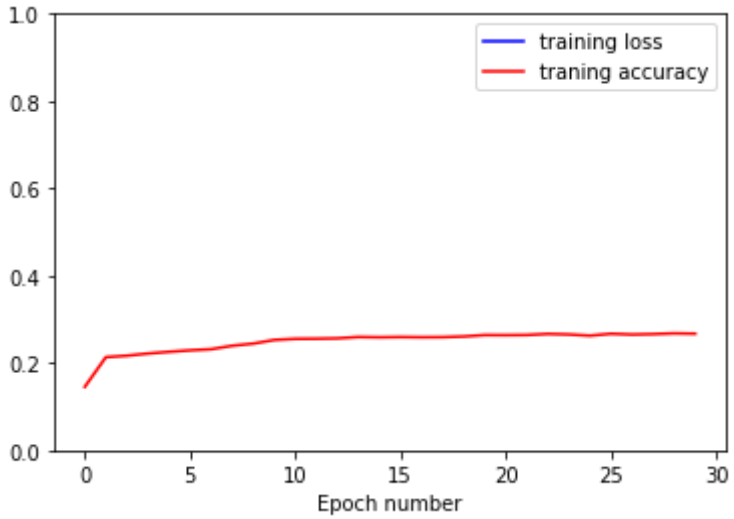
## num_hidden =[1,1,1], Epochs = 30, batch_size = 2, AF = sigmoid

number of layers: 4

Total params: 809

Trainable params: 809

Non-trainable params: 0




final training loss: 1.750355

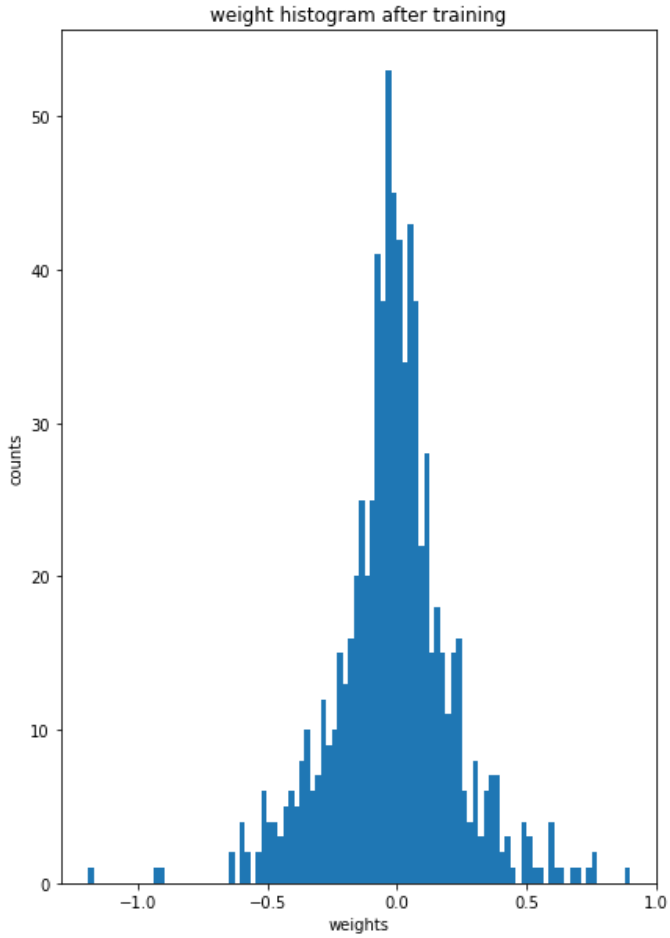
final training accuracy: 0.267160

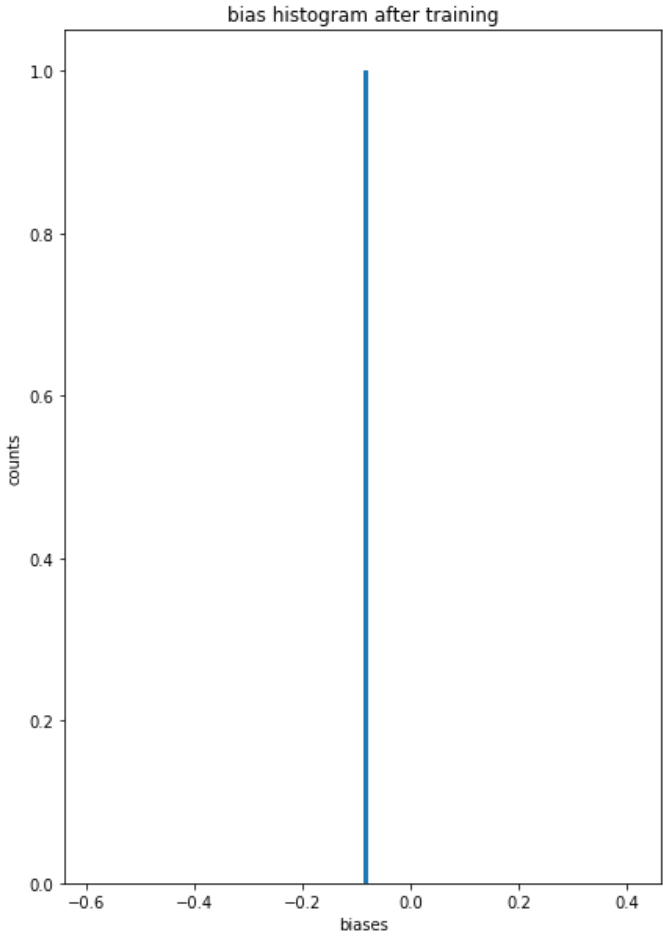
final validation loss: 1.760427

final validation accuracy: 0.242700

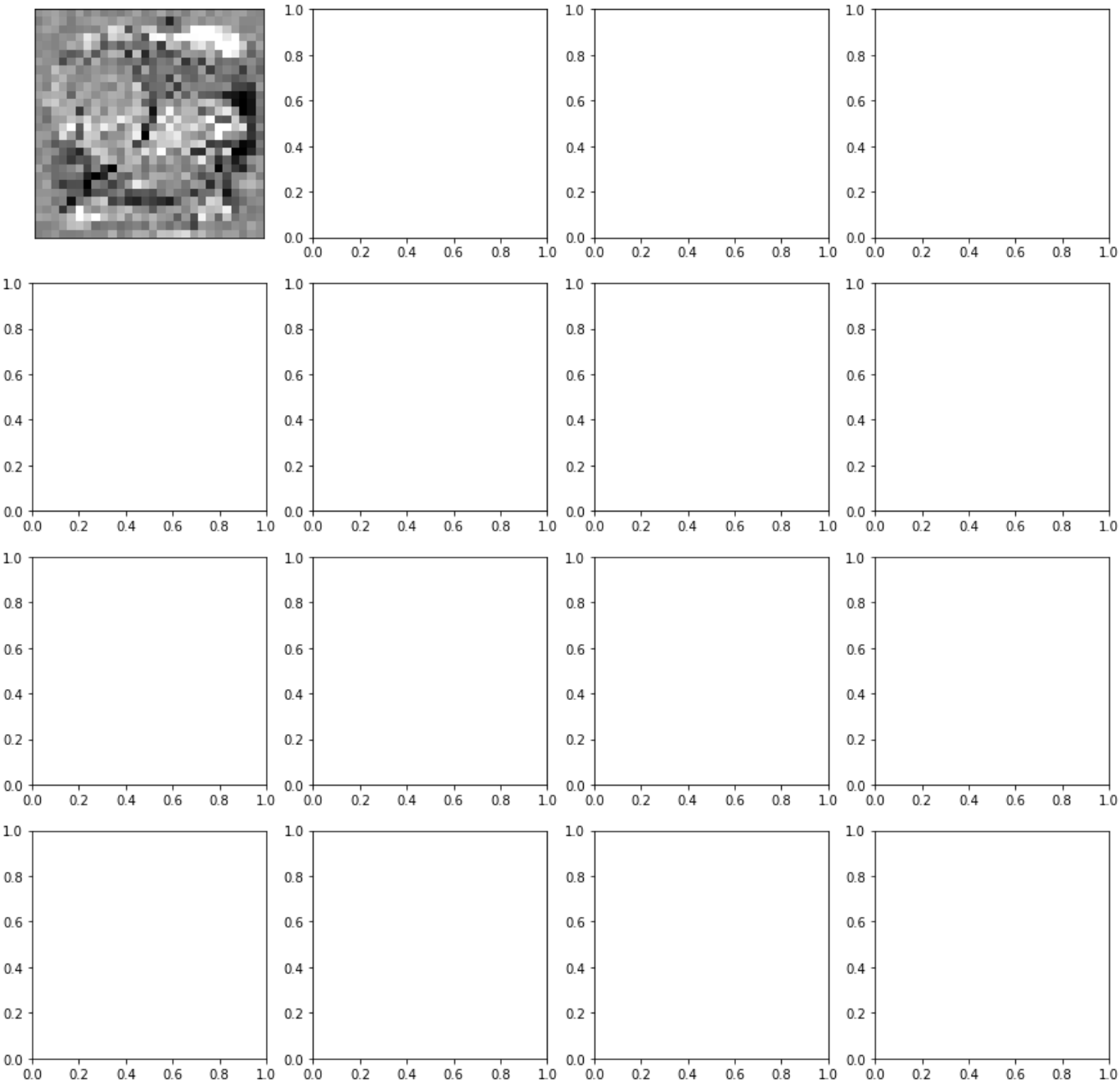
final test loss: 1.769383

final test accuracy: 0.241900




Visualization of the weights between input and some of the hidden neurons of the first hidden layer:



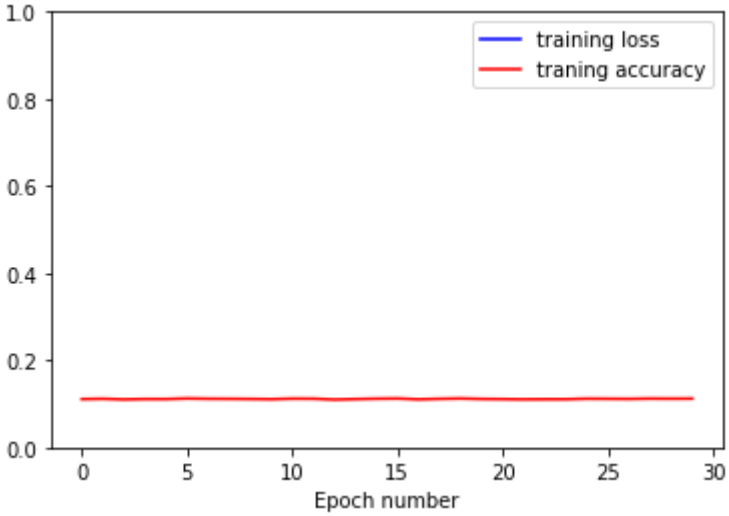
# num_hidden =[1,1,1], Epochs = 30, batch_size = 2, AF = relu

... number of layers: 4

Total params: 809

Trainable params: 809

Non-trainable params: 0




final training loss: 2.302084

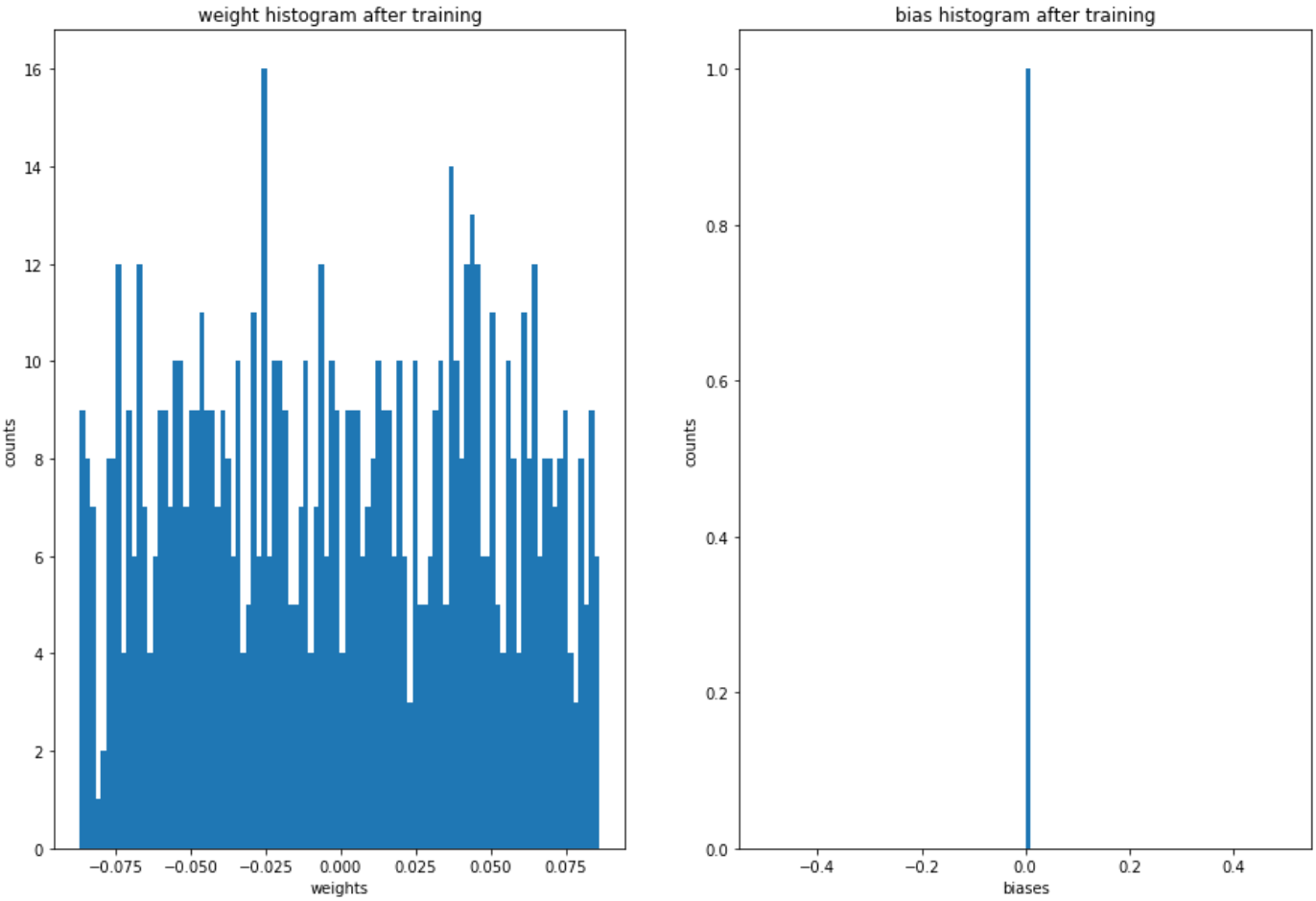
final training accuracy: 0.112120

final validation loss: 2.304083

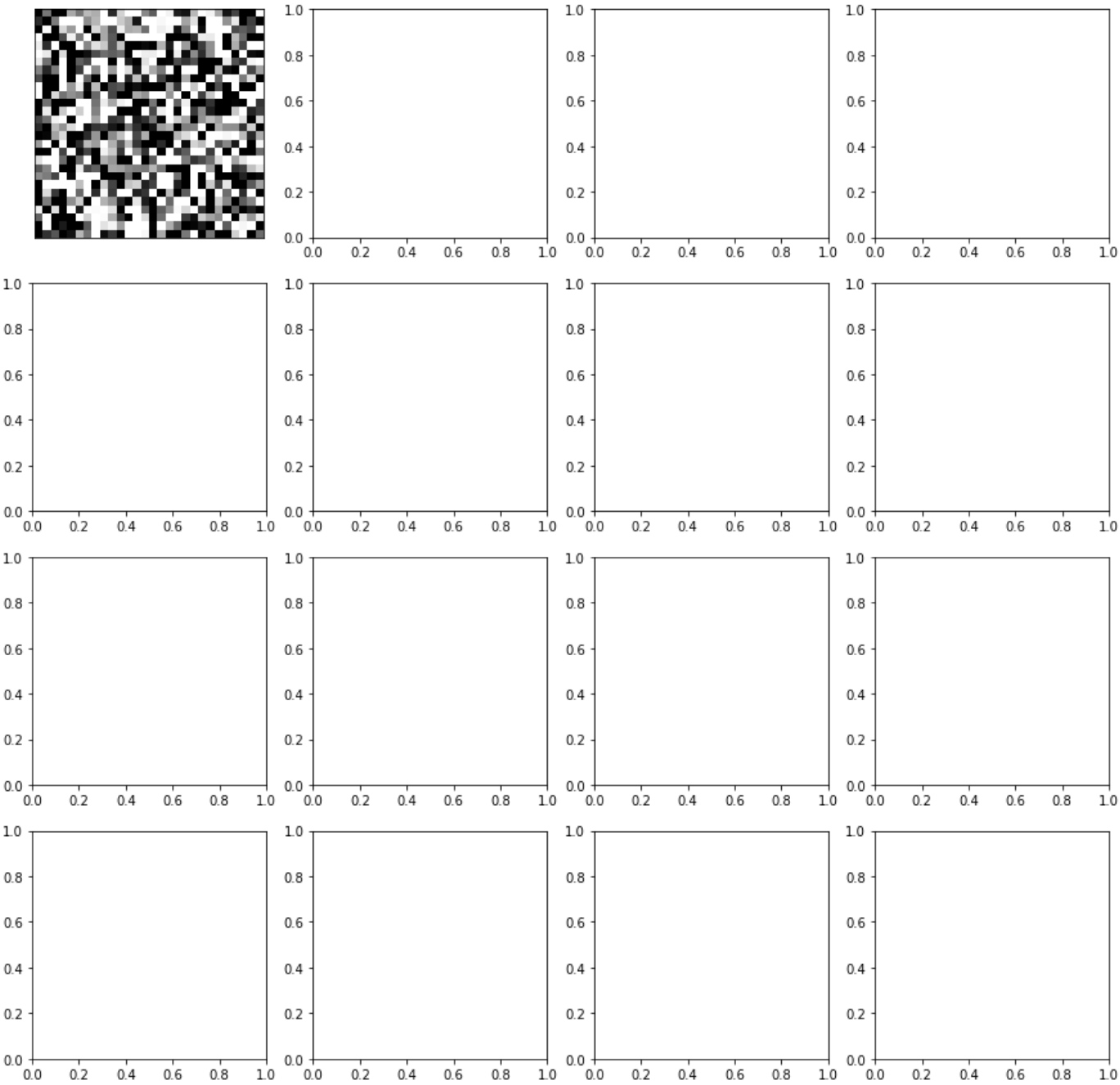
final validation accuracy: 0.106400

final test loss: 2.302129

final test accuracy: 0.113500




Visualization of the weights between input and some of the hidden neurons of the first hidden layer:



## Findings

| | # hidden layer 3 (sigmoid) | # hidden layer 3 (relu) |
|---|---|---|
| final training loss | 1.750355 | 2.302084 |
| final training accuracy | 0.267160 | 0.112120 |
| final validation loss | 1.760427 | 2.304083 |
| final validation accuracy | 0.242700 | 0.106400 |
| final test loss | 1.769383 | 2.302129 |
| final test accuracy | 0.241900 | 0.113500 |

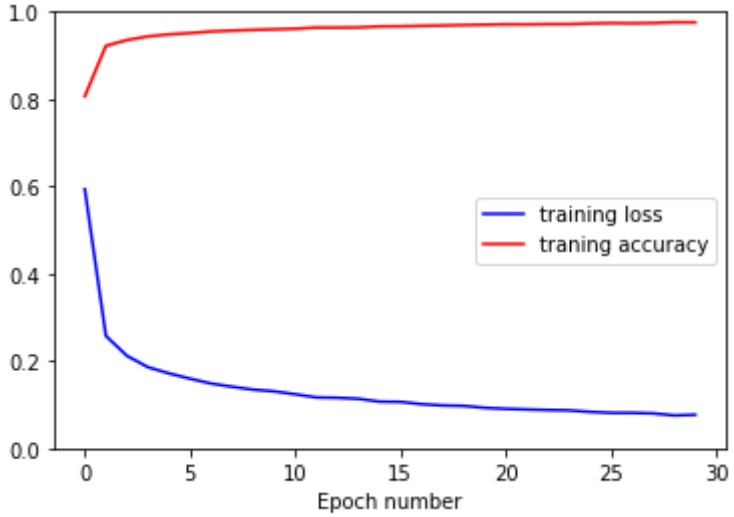
In short, with less number of perceptron sigmoid acivation function works better.

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.

# Answer

### num_hidden = [15,25,50,75,100,150], Activation = relu, learningRate= 0.01, batch = 4, epochs = 30, regularization_weight = 0.0




final training loss: 0.076854

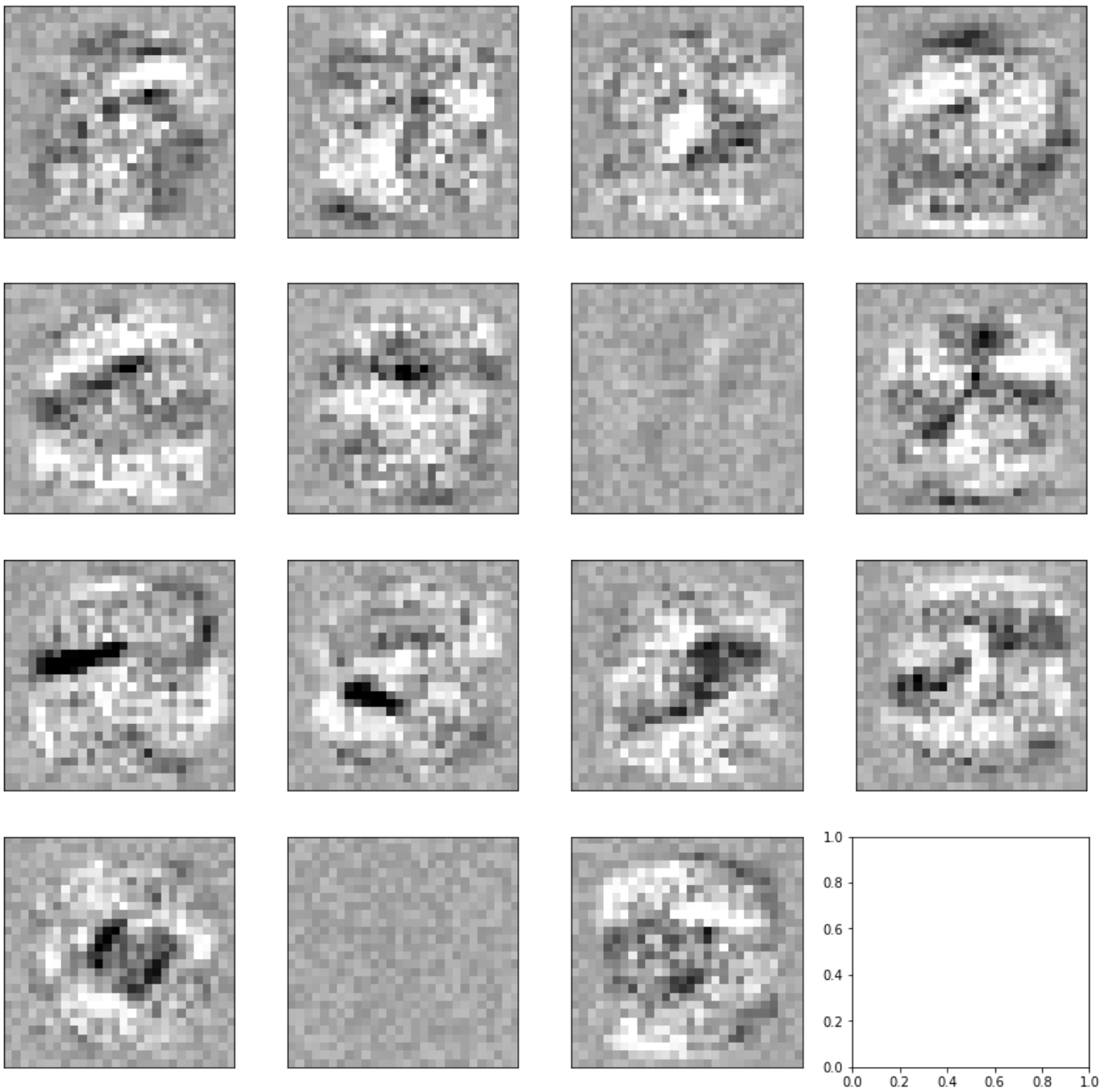
final training accuracy: 0.975940

final validation loss: 0.181777

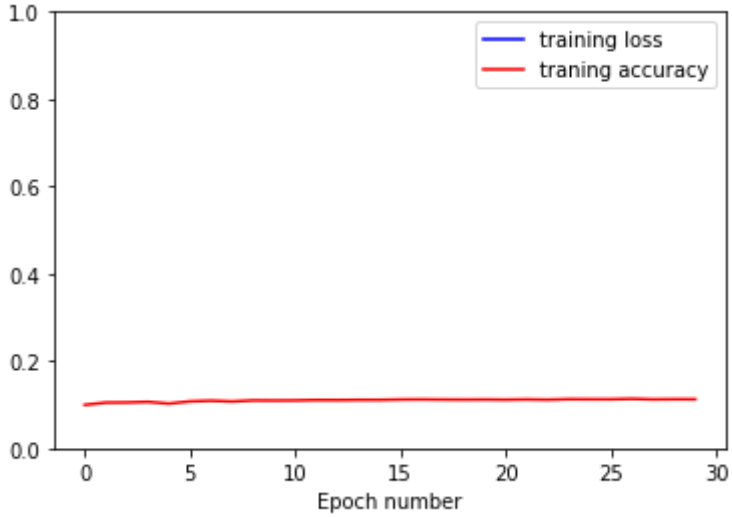
final validation accuracy: 0.953900

final test loss: 0.203929

final test accuracy: 0.950300




**num_hidden = [15,25,50,75,100,150], Activation = sigmoid, learningRate= 0.01, batch = 4, epochs = 30, regularization_weight = 0.0**



final training loss: 2.301880

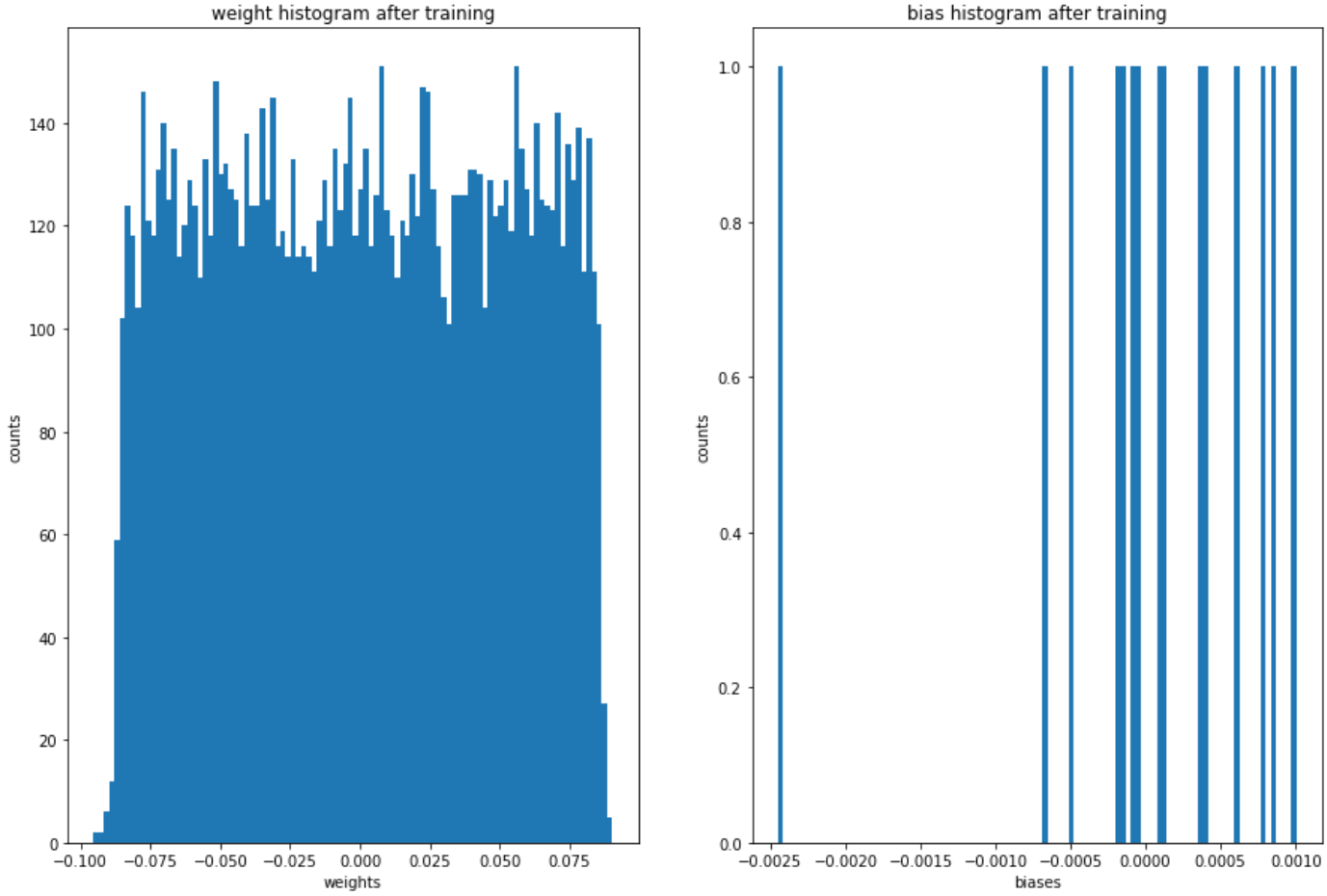
final training accuracy: 0.112420

final validation loss: 2.304394

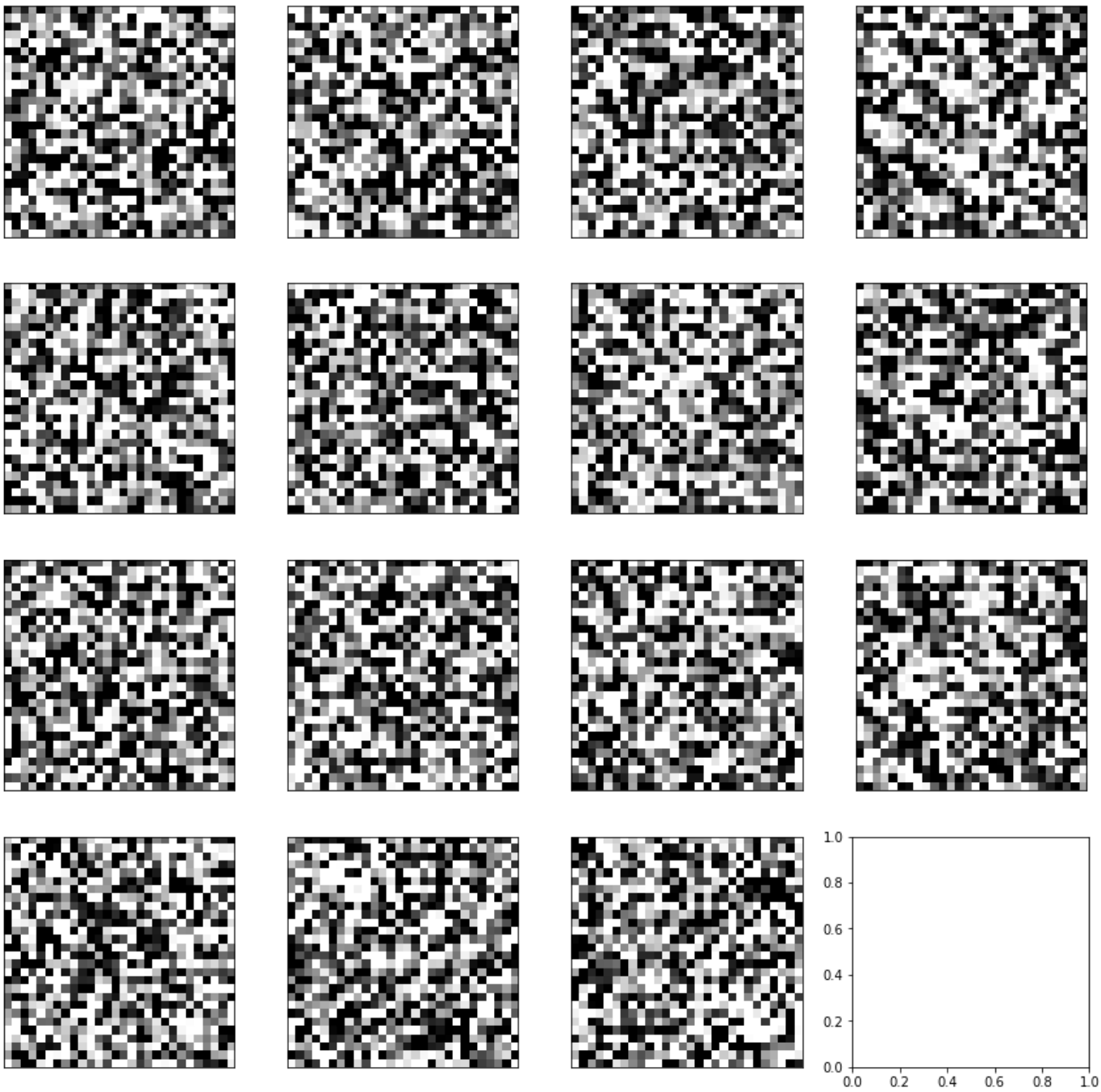
final validation accuracy: 0.106400

final test loss: 2.302055

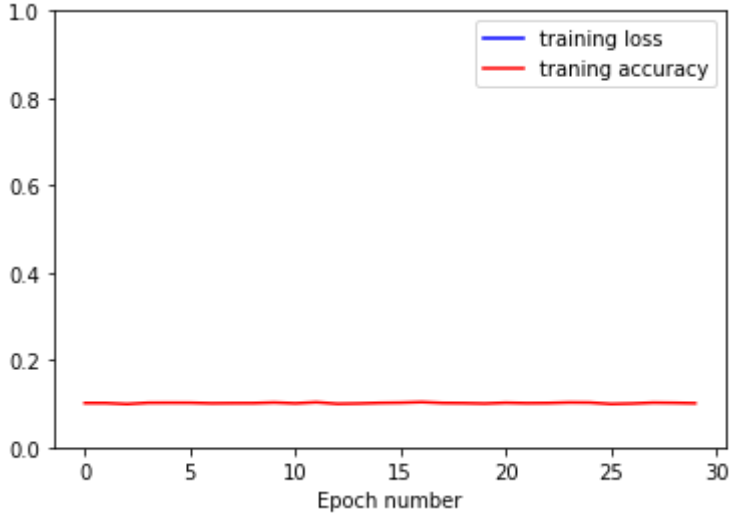
final test accuracy: 0.113500




Visualization of the weights between input and some of the hidden neurons of the first hidden layer:




**num_hidden = [15,25,50,75,100,150], Activation = sigmoid, learningRate= 0.01, batch = 4, epochs = 30, regularization_weight = 0.01, dropout = 0.2**



final training loss: 2.374155

final training accuracy: 0.100940

final validation loss: 2.358470

final validation accuracy: 0.106400

final test loss: 2.357207

final test accuracy: 0.113500

## num_hidden = [16,25,50,75,100,150], initialLearningRate = 0.001, batch_size = 4, dropout = 0.0, regularization_weight = 0.0, batch_normalization = False

final training loss: 2.303313

final training accuracy: 0.108400

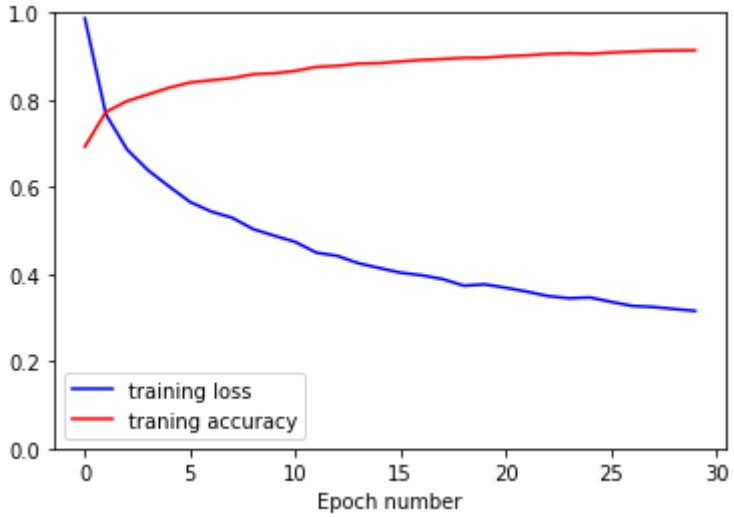
final validation loss: 2.303749

final validation accuracy: 0.106400

final test loss: 2.302264

final test accuracy: 0.113500

## num_hidden = [16,25,50,75,100,150], Activation = sigmoid, initialLearningRate = 0.01, batch_size = 4, dropout = 0.0, regularization_weight = 0.0, batch_normalization = True




final training loss: 0.315673

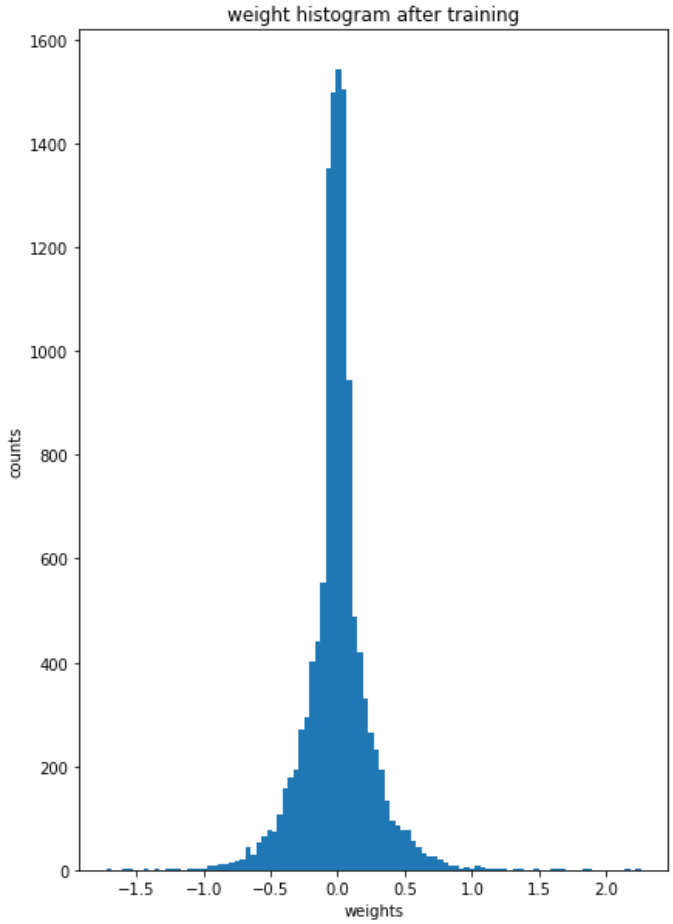
final training accuracy: 0.913880

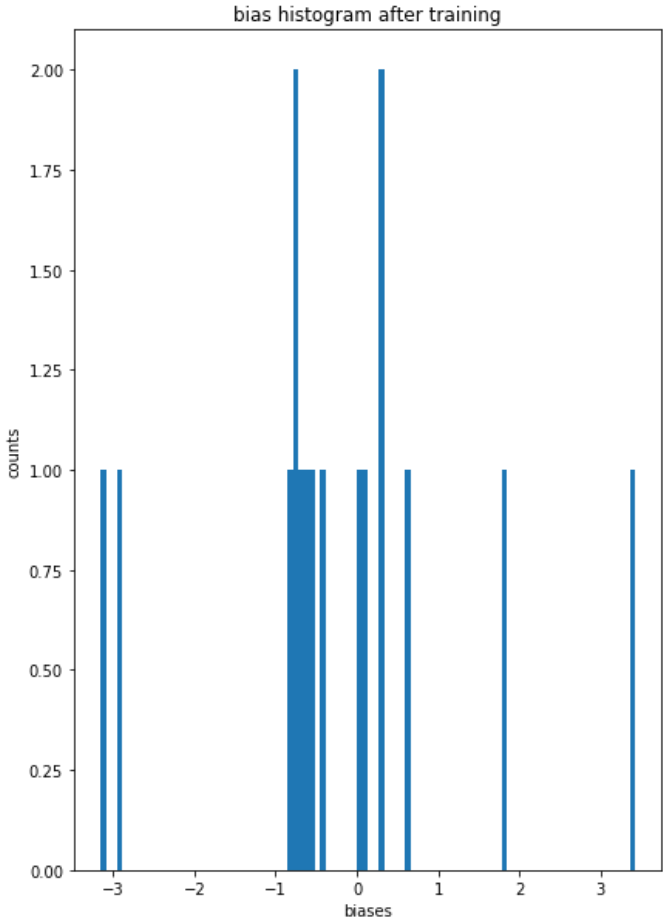
final validation loss: 0.813610

final validation accuracy: 0.724200

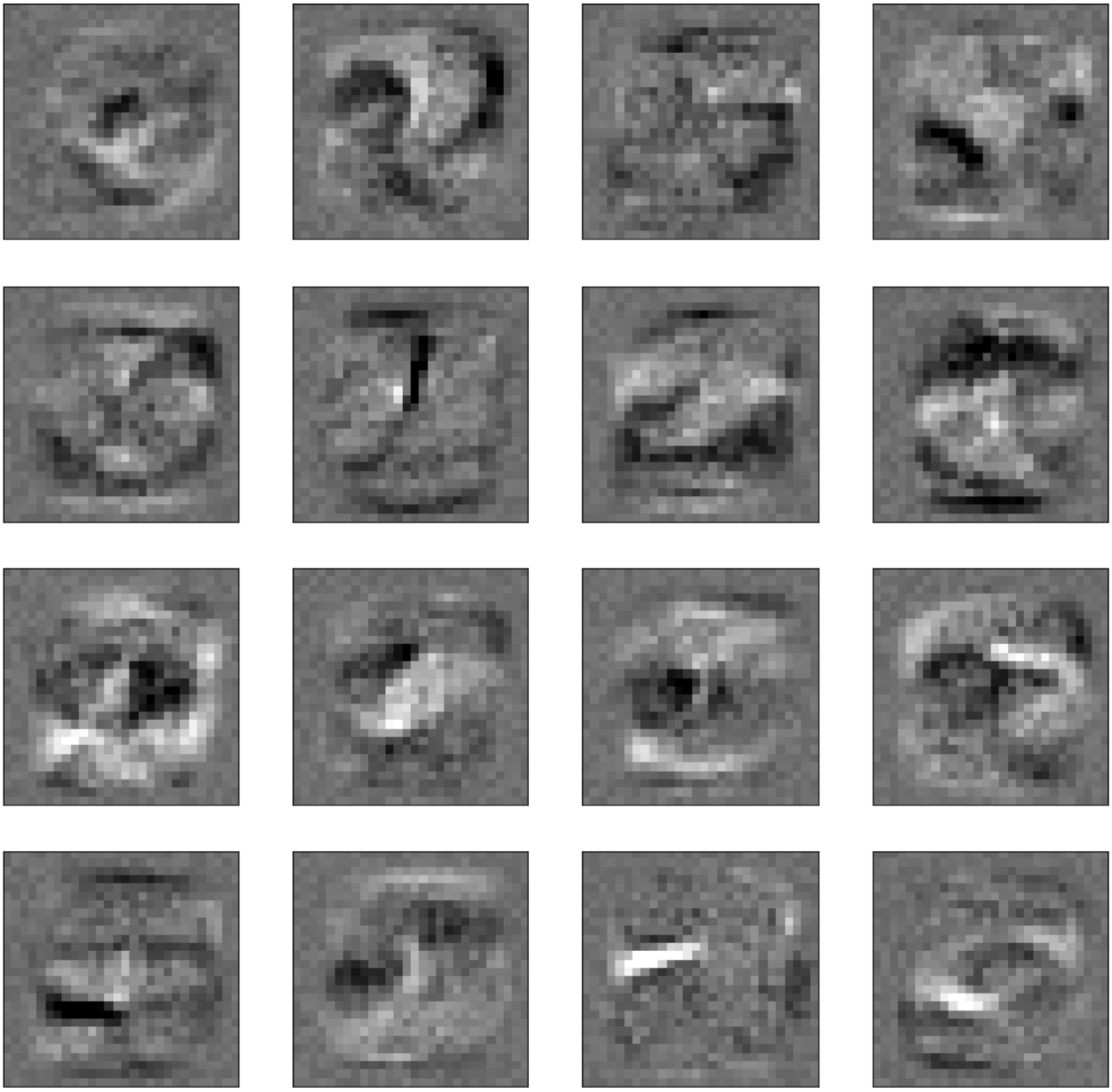
final test loss: 0.832318

final test accuracy: 0.708700




Visualization of the weights between input and some of the hidden neurons of the first hidden layer:



## num_hidden = [16,25,50,75,100,150], Activation = relu, initialLearningRate = 0.01, batch_size = 4, dropout = 0.0, regularization_weight = 0.0, batch_normalization = True

final training loss: 0.315673

final training accuracy: 0.913880

final validation loss: 0.813610

final validation accuracy: 0.724200

final test loss: 0.832318

final test accuracy: 0.708700

**Findings**

Training accuracy always increase if we use "relu" activation function with large number of perceptron each layer.