For lab 1 you implemented the forward or inference part of a feedforward neural network and tested it using the MNIST dataset. For this lab you will experiment with ways of learning the weights in the network that allow it to classify digits accurately. For this lab you will use the same test sets used in the previous lab, and you will also need a training set, included in files `xtrain.txt` and `ytrain.txt`.

Implement the following algorithms and compare their performance in terms of accuracy and training time. Feel free to try different network architectures and parameters. You may consider using a single hidden layer, or hidden layers with fewer units.

**Algorithm 1: Random search**

Generate random sets of weights, evaluate them on a subset of the training data, and keep the best set of weights found after a number of iterations.

```python
findWeightsRandom(X,Y)
let W_best = ⟨W_0,W_1,W_2,b_0,b_1,b_2⟩ be a randomly-chosen set of weights
for i = 1:numreps
  let ⟨X_b,Y_b⟩ be a randomly chosen subset of ⟨X,Y⟩
  W = W_best + ∆W, where ∆W is a randomly-chosen set of weight changes
  e_0 = ∑((Y_b - ff(W,X_b))^2)
  e_1 = ∑((Y_b - ff(W_best,X_b))^2)
  if e_0 < e_1
    W_best = W
return W_best
```

Here $ff(W,X)$ is the output of the feedforward neural network (as implemented in lab 1) using weights $W$ and input data $X$ and $\sum(Y)$ is the sum the elements of matrix or vector $Y$. Randomly chosen weights should be obtained from a distribution with zero mean; use for example `numpy.random.randn(rows,cols)-0.5)*s`, where $s$ is a scale factor. This would produce random numbers uniformly distributed from in the $[-s/2,s/2]$ range. I have used values of $s$ around 0.2 for the initial value of $W$ and somewhat smaller for $\Delta W$. $⟨X_b,Y_b⟩$ the randomly chosen subset of the training set, is known as a batch. Appropriate batch sizes range from 50 to 1000. A new batch is chosen for every iteration of the algorithm.

**Algorithm 1.5: Linear model without hidden layers**

Implement a linear model $P = xW + b$ using the update equations described in class.

**Algorithm 2: Random initial layers, last layer computed directly from data**

Generate random sets of weights for the first two layers, then compute the weights in the last layer using least-squares approximation.

```python
findWeightsPseudoinverse(X,Y)
  Choose ⟨W_0,W_1,b_0,b_1⟩ randomly
  H_0 = relu(XW_0 + b_0)
  H_1 = relu(H_0W_1 + b_1)
  Y_L = logit(Y)
  b_2 = µ(Y_L)
  W_2 = H_1^T(Y_L - b_2)
returns ⟨W_0,W_1,W_2,b_0,b_1,b_2⟩
```
Here $A^+$ is the Moore-Penrose pseudo-inverse of matrix $A$. Use the python function `numpy.linalg.pinv` to compute $A^+$. The $\logit$ function is the inverse of the $\sigma$ function, $\logit(x) = \ln(x) - \ln(1-x)$. Also, do the transformation $Y = Y \ast 0.9 + 0.05$ before training to keep the arguments of the $\logit$ function in a valid range.

**Algorithm 3: Gradient descent and backpropagation**

This is the classical calculus-derived algorithm for optimizing weights. It makes extensive use of the chain rule to find the gradients of the network error with respect to the weights in the input and hidden layers.

```python
def findWeightsBackprop(X, Y):
    W = \{W_0, W_1, W_2, b_0, b_1, b_2\} be a randomly-chosen set of weights
    numreps = 1
    for i = 1:numreps
        let (X_b, Y_b) be a randomly chosen subset of \{X, Y\}
        Compute forward pass
        $H_0 = \text{relu}(X_0 W_0 + b_0)$
        $H_1 = \text{relu}(H_0 W_1 + b_1)$
        $P = \sigma(H_1 W_2 + b_2)$
        Compute error gradient for last layer
        $\Delta H_1 = (P - Y_b) \odot P' = (P - Y_b) \odot P \odot (1 - P)$
        Compute error gradient for hidden layers
        $\Delta H_0 = (\Delta P W_1^T) \odot H_1' = (\Delta P W_1^T) \odot \text{sign}(H_1)$
        $\Delta P = (\Delta P W_1^T) \odot H_1' = (\Delta P W_1^T) \odot \text{sign}(H_0)$
        Update weights and biases
        $W_2 = W_2 - \lambda H_1^T \Delta P$
        $W_1 = W_1 - \lambda H_0^T \Delta H_1$
        $W_0 = W_0 - \lambda X_0^T \Delta H_0$
        $b_2 = b_2 - \lambda \sum(\Delta P)$
        $b_1 = b_1 - \lambda \sum(\Delta H_1)$
        $b_0 = b_0 - \lambda \sum(\Delta H_0)$
    return (W_0, W_1, W_2, b_0, b_1, b_2)
```

Here $\sum(X)$ is a vector, the column-wise sum of $X$ and $\lambda$ is the learning rate, a small constant (values around 0.001 are appropriate). As before, do the transformation $Y = Y \ast 0.9 + 0.05$, as the sigmoid function can only output values of 0 or 1 when the argument is $-\infty$ or $\infty$, respectively.

**Algorithm 4: Combine algorithms 2 and 3**

Algorithm 4 is identical to Algorithm 3, except that instead of randomly initializing the weights, it uses Algorithm 2 to find the initial value of $W$.

**Notes**

Algorithms 1, 3 and 2 are easy to implement and should provide reasonable results, with accuracies on the test set of about 0.50, 0.85 and 0.90, respectively. Algorithm 3 is harder to program and debug. One way to check if your program is making accurate progress is to display the mean of the error function $(\mu(Y_b - P)^2)$ and/or the accuracy after every learning iteration. If the error is not decreasing, you probably have a bug. Another possibility is to allow learning only in one layer at a time, starting with the last layer, until you make sure your program is working. Thus, at first you only update $W_2$ and $b_2$, keeping $W_0, W_1, b_0, b_1$ constant. Once you see the error is decreasing, you test keeping $W_0, W_2, b_0, b_2$ constant and updating $W_1$ and $b_1$, and so on. Algorithm 3 should yield an accuracy of around 97% on the test set. Algorithm 4 should yield similar results, but require fewer training iterations. For all the algorithms, $Y$ is the one-hot representation of the label (not the label itself).

As before, write a report describing your work. Include plots of the error function as training progresses (except for algorithm 2, which is not iterative). Comment on the relative performances of the algorithms. Also, include tables illustrating the running times and accuracies (on the test set) of the algorithms. Include your code as an appendix.