Neural Nets, continued

0.1 Post Training Analysis

How do we determine whether or not our network is doing a good job? Below we consider some options for testing the neural network.

1. We should always break our data into a training set and a testing set. We can report the error on the testing set as we've discussed earlier.

2. Ideally the error that we do get should be close to our desired output, and there should be no patterns in the error.

   (a) We can perform a linear regression on between the target values and the neural output. If the targets are multidimensional, then we can do this for each coordinate to see if the neural output is close to the desired output- in fact, we can compute the correlation between the targets and the neural outputs- we should be getting numbers close to 1. Figure 1 shows a typical kind of ouput.

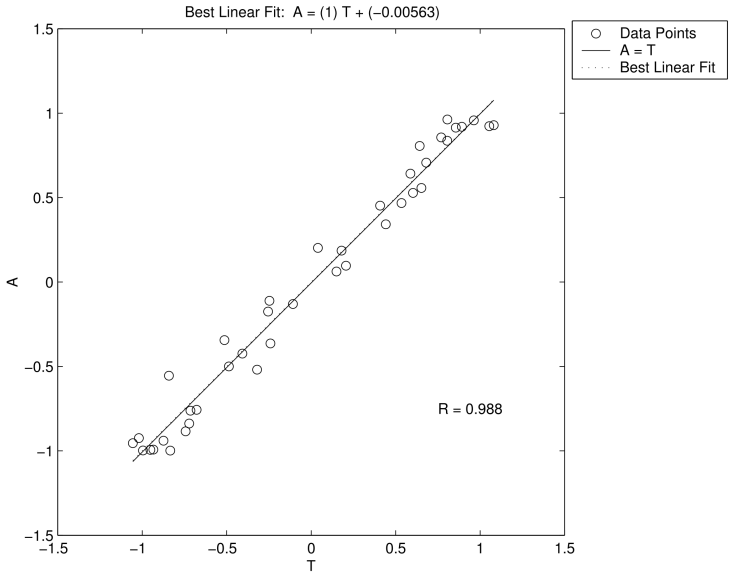


Figure 1: The result of using post regression on the targets. Obtaining a correlation as shown means that the network has trained well.

   (b) A second method is taken from the idea that if the training is good, then the error should be uncorrelated. Put another way, all correlations should be explained by the neural network model. We can therefore look at the correlations in the error. For example, uppose matrix T is 3×500

(500 points in $\mathbb{R}^3$), and matrix Y is our neural output that is 3×500. If our training was good, the error data, $T - Y$ should be uncorrelated in $\mathbb{R}^3$. In Matlab, we could compute the SVD of the error and check the singular values to see that they are all close to zero.

```
G=T-Y;
[U,S,V]=svd((1/500)*G*G');
S=
   1.0e-06 *

    0.3983         0         0
         0    0.3534         0
         0         0    0.0042
```

(c) The last method we discuss is similar to the second. We can examine the histogram of the size of the errors. If training went well, we can expect the errors to be clustering about zero, and rapidly trail off.

3. Examine the weights of the network. Overtraining will generally lead to very large weights (recall what some of the weights looked like in the RBF network). Extremely large weights are undesirable, because they can lead to instabilities in the network- i.e., small deviations in the domain can lead to huge deviations in the range.

   Similarly, weights that are extremely small may suggest that training could have been done with fewer nodes in the hidden layer.

4. Regularization. A detailed explanation of regularization is beyond the scope of these notes, but we can give some heuristics. The basic idea is to penalize those weights that get very large in favor of small weights. Formally, we change the error function. Let E denote our standard mean squared error. Assume all training parameters (weights and biases) are indexed as: W_i, $i = 1, 2, \ldots, N$. The new error function, MSE_{reg}, is defined as:

$$\text{MSE}_{\text{reg}} = \gamma E + (1 - \gamma)\frac{1}{N}\sum_{i=1}^{N} W_i$$

   where γ is called the *regularization parameter*. As we can see, if $\gamma = 1$, the weights play no role in the training, but if $\gamma = 0$, the weights play the only role in the training. In general, the smaller the parameter, the more smooth the approximation.

   The problem is in knowing what to make the regularization parameter. If it is too large, the network may overfit. If it is too small, we may not be modeling the data very well.

These are just a few of the choices one may use in post training. It takes practice to know which works best under which conditions.

0.2 Example: Handwritten Digits Recognitioon

We're already familiar with this data (or something similar), so this will be a good way to compare methods.

The letters we looked at before were something like 28×28, but this time (since we're doing things by hand), we'll look at numbers represented by a 5×7 matrix of values that are either 1 or 0. Therefore, the domain of the neural network will be 35 dimensional.

The output of the neural network will be 26-dimensional, where there is a 1 in the numerical position corresponding to its letter.

Our goal: Character recognition.

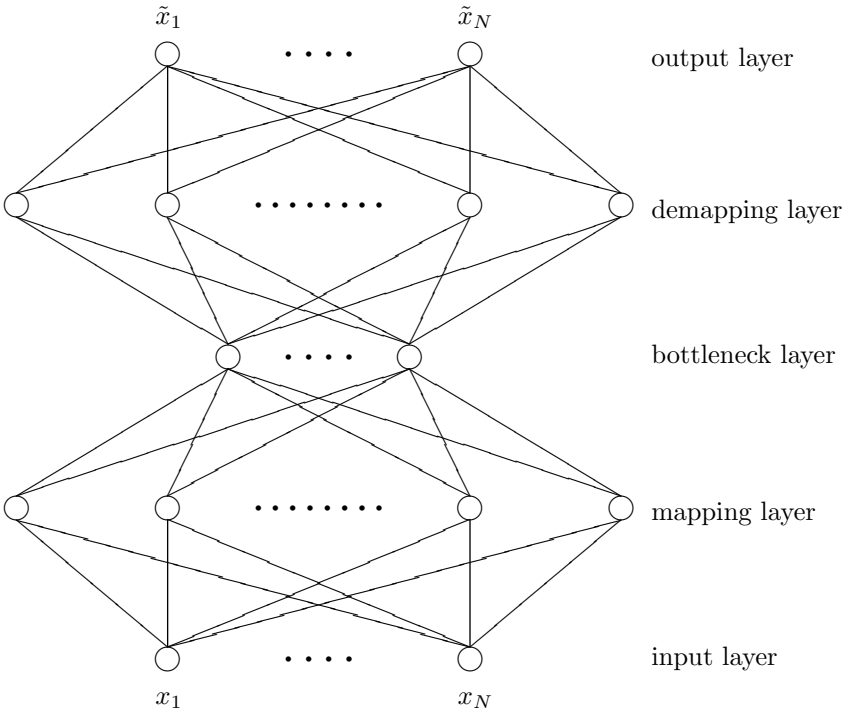


Figure 2: The bottleneck network architecture.

Architectural Items

- We will utilize a $35 - 10 - 26$ feed forward neural network using a `logsig` transfer function for both the hidden and output layers.
- We will use a standard gradient descent for training (this can be slow). Let's use a momentum term with $\beta = 0.9$, and a maximum of 5000 epochs.
- At the end, since this is a classification problem, we would want to check the confusion matrix (although it will be large).

0.3 Autoassociation Neural Networks, or Autoencoders

In the previous section, we saw that a three layer feedforward neural network is capable of approximating any continuous function on a compact domain.

Suppose now that we want to reduce the dimensionality of the data set X. This can be expressed as constructing homeomorphisms

$$G : X \subset \mathbb{R}^N \to Y \subset \mathbb{R}^M$$
$$H : Y \subset \mathbb{R}^M \to X \subset \mathbb{R}^N$$

We could build two separate neural networks, one for each function, if we knew a priori the target values in the reduced dimensional space Y. For the moment, suppose we are not concerned with the form of the reduced representation. Then the compression paradigm can be reformulated: find functions G and H as before, but we only require that the composition $G \circ H$ is the identity on X. We envision this composition as a single, large network composed of two, three layer networks glued together, as shown in Figure 2.

In this figure, the input and output layers have N nodes, the bottleneck layer has $M << N$ nodes, and the two hidden layers have some pre-determined number of nodes. It has been shown in [**?**, **?**, **?**] that

these hidden layers are necessary to perform a nonlinear compression and decompression. That is, without the nonlinear hidden layer, this network performs a linear dimensionality reduction- for which we know the optimal solution is given by the principal components in PCA (SVD).

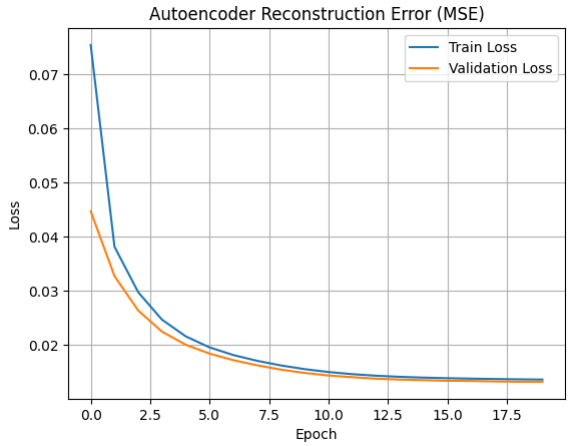
Coding the Autoassociator

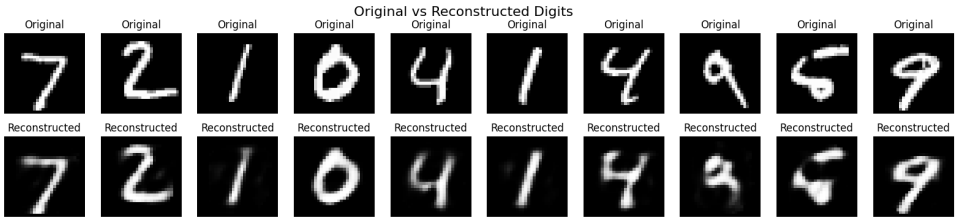
Here's an example of creating a simple autoencoder in Python. We use the MNIST digit data ("digits" are 28×28 arrays, which are concatenated to vectors in $\mathbb{R}^{784}$). In this case, we'll put 20 nodes in the bottleneck layer.

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam

# Load MNIST data
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Concatenate or flatten the input
#   from 28 x 28 to vectors with 784 elements
x_train = x_train.reshape((x_train.shape[0], 784))
x_test = x_test.reshape((x_test.shape[0], 784))

# Define the autoencoder architecture
input_img = Input(shape=(784,))
encoded = Dense(25, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input_img, decoded)

# Compile the model
autoencoder.compile(optimizer=Adam(), loss='mse')

# Train the autoencoder
history = autoencoder.fit(x_train, x_train,
                          epochs=20,
                          batch_size=256,
                          shuffle=True,
                          validation_data=(x_test, x_test))

# Plot training and validation loss
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title("Autoencoder Reconstruction Error (MSE)")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
```

```
plt.show()

# Reconstruct some digits
decoded_imgs = autoencoder.predict(x_test)

# Visualize original vs reconstructed digits
n = 10  # Number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.axis('off')

    # Reconstructed
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    plt.title("Reconstructed")
    plt.axis('off')
plt.suptitle("Original vs Reconstructed Digits", fontsize=16)
plt.show()
```

And the output that we see:





It is easy to add another hidden layer between the input and bottleneck and between the bottleneck layer and the output. Here's an example:

```
# Define the autoencoder architecture
input_img = Input(shape=(784,))
input_hidden = Dense(75,activation='relu')(input_img)
encoded = Dense(25, activation='relu')(input_hidden)
```

```
decoded_hidden=Dense(75,activation='relu')(encoded)
decoded = Dense(784, activation='sigmoid')(decoded_hidden)

autoencoder = Model(input_img, decoded)
```

Why use an autoencoder?

The autoencoder is an excellent example of how you could run your neural network in reverse. Remember that the input to the network is an image, and the bottleneck is a vector in $\mathbb{R}^2 5$. Therefore, for each vector we come up with in $\mathbb{R}^{25}$, we should be able to find a corresponding image.

Remember that the goal of generative adversarial neural networks is to create new data that the neural net has not seen, but that is very similar to the data being input. You might be able to imagine that, instead of a simple vector of length 784, we were working with high resolution images of people, or animals, or landscapes. Then the autoencoder gives you a way of generating these new images from vectors in the bottleneck.

In practice, the bottleneck space does not smoothly encode the input space. In other words, images with the digit "2" may not all be clustered nicely together in the bottleneck space, and in fact pieces may be scattered throughout the 25 dimensional space. We have to have a way of enforcing some kind of smoothness parameter to the bottleneck layer- and that brings us to Variational Autoencoders. These are the autoencoders that can be used to generate new data for the adversarial networks. It would take us too far afield to go into detail on how this is done, but you should explore what's been done in the past 5-10 years.