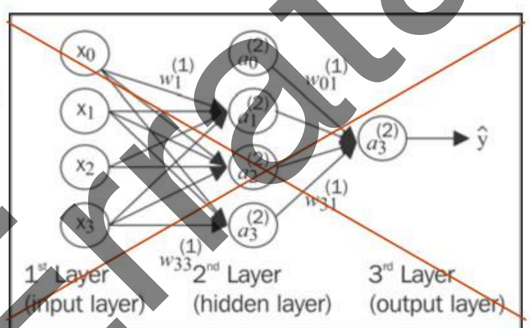
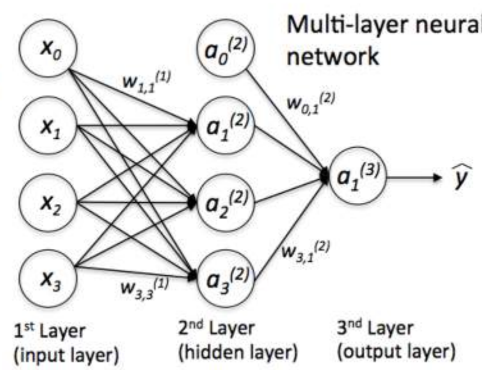
> Note that although Adaline consists of two layers, one input layer and one output layer, it is called a single-layer network because of its single link between the input and output layers.

Introducing the multi-layer neural network architecture

In this section, we will see how to connect multiple single neurons to a **multi-layer feedforward neural network**; this special type of network is also called a **multi-layer perceptron** (**MLP**). The following figure explains the concept of an MLP consisting of three layers: one input layer, one hidden layer, and one output layer. The units in the hidden layer are fully connected to the input layer, and the output layer is fully connected to the hidden layer, respectively. If such a network ha
hidden layer, we also call it a *deep* artificial neural network.

Unfortunately, a lot of typos were made in this figure, please refer to my original to the right





> We could add an arbitrary number of hidden layers to the MLP to create deeper network architectures. Practically, we can think of the number of layers and units in a neural network as additional **hyperparameters** that we want to optimize for a given problem task using the cross-validation that we discussed in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.
>
> However, the error gradients that we will calculate later via backpropagation would become increasingly small as more layers are added to a network. This *vanishing gradient* problem makes the model learning more challenging. Therefore, special algorithms have been developed to pretrain such deep neural network structures, which is called *deep learning*.

```
            grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))

            return grad1, grad2

        def predict(self, X):
            a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
            y_pred = np.argmax(z3, axis=0)
            return y_pred

        def fit(self, X, y, print_progress=False):
            self.cost_ = []
            X_data, y_data = X.copy(), y.copy()
            y_enc = self._encode_labels(y, self.n_output)

            delta_w1_prev = np.zeros(self.w1.shape)
            delta_w2_prev = np.zeros(self.w2.shape)

            for i in range(self.epochs):

                # adaptive learning rate
                self.eta /= (1 + self.decrease_const*i)

                if print_progress:
                    sys.stderr.write(
                            '\rEpoch: %d/%d' % (i+1, self.epochs))
                    sys.stderr.flush()

                if self.shuffle:
                    idx = np.random.permutation(y_data.shape[0])
                    X_data, y_data = X_data[idx], y_data[idx]

                mini = np.array_split(range(
                             y_data.shape[0]), self.minibatches)
                for idx in mini:

                    # feedforward
                    a1, z2, a2, z3, a3 = self._feedforward(
                                        X[idx], self.w1, self.w2)
                    cost = self._get_cost(y_enc=y_enc[:, idx],
                                          output=a3,
                                          w1=self.w1,
                                          w2=self.w2)
                    self.cost_.append(cost)
```

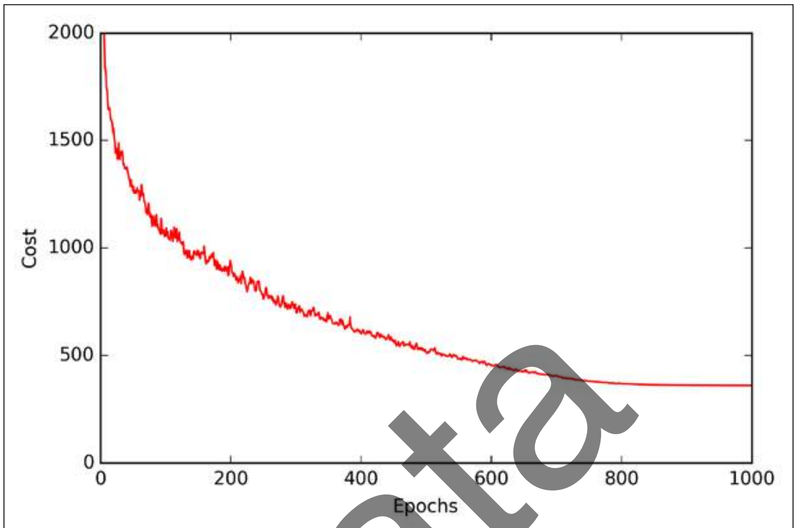
`X_data, y_enc = X_data[idx],  y_enc[:,idx]`

`X_data[idx]`


```
>>> nn = NeuralNetMLP([...],
...                   [...],
...                   shuffle=False,
...                   random_state=1)
```

These line changes above enable shuffling if the setting is `shuffle=True`.
To match the original output in the book (no shuffling) after applying this patch, the `shuffle=False` setting needs to be added when the NeuralNetMLP is initialized (next page) as shown on the left.

The following plot gives us a clearer picture indicating that the training algorithm converged shortly after the 800th epoch:



Now, let's evaluate the performance of the model by calculating the prediction accuracy:

```
>>> y_train_pred = nn.predict(X_train)
>>> acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Training accuracy: 97.74%
```

As we can see, the model classifies most of the training digits correctly, but how does it generalize to data that it has not seen before? Let's calculate the accuracy on 10,000 images in the test dataset:

Test

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
>>> print('Training accuracy: %.2f%%' % (acc * 100))
Test accuracy: 96.18%
```

Based on the small discrepancy between training and test accuracy, we can conclude that the model only slightly overfits the training data. To further fine-tune the model, we could change the number of hidden units, values of the regularization parameters, learning rate, values of the decrease constant, or the adaptive learning using the techniques that we discussed in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning* (this is left as an exercise for the reader).