Deep Learning Report

April 2023

Dana Aubakirova
Université Paris-Saclay

Stefano Bavaro
Université Paris-Saclay

Benedictus Kent Rachmat
Université Paris-Saclay

Diego Andres Torres Guarin
Université Paris-Saclay

I. Introduction

In recent years, deep learning has revolutionized the field of artificial intelligence by enabling machines to learn from vast amounts of data and make decisions with increasing accuracy. One of the reasons behind the widespread adoption of neural networks is the introduction of deep learning libraries that make the process of training these models much more accessible. These libraries allow engineers and researchers to focus on the high-level design of the network architecture and objective function, while the library takes care of the details of computing gradients and performing backpropagation.

However, one potential drawback of using automatic differentiation libraries is that users may not fully understand how the libraries work under the hood. In particular, it can be challenging to understand how the gradients are computed and how they propagate through the network during training. This can lead to subtle bugs and numerical instabilities that are difficult to detect and diagnose. Therefore, it is important to have a solid understanding of the underlying mathematics and algorithms behind deep learning.

In this report we will explain the details and mathematical foundations of our implementation of a small deep learning library. We will focus on the most widely known architecture, the feed forward neural network (shown in figure 1) A feed-

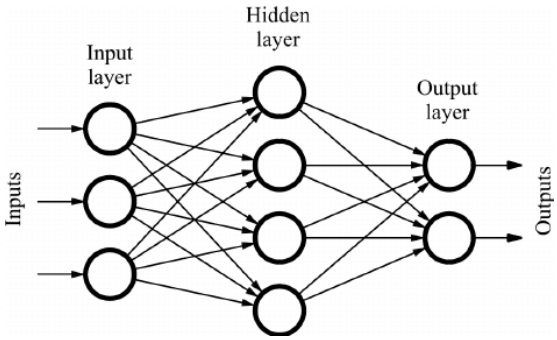

Fig. 1. Diagram of a feed forward neural network.

forward neural network is a type of artificial neural network where the flow of information moves only in one direction, from the input layer, through the hidden layers, and finally to the output layer. The input layer is simply the layer that contains the values of the input data, represented as vectors of a fixed dimension. Each hidden layer consists of a set of neurons that apply affine transformations, which are essentially a weighted sum of the inputs plus a bias term. In mathematical notation, if x is the input to the layer (probably coming from a previous hidden layer), then the value of the output y is given by

$$y = Wx + b, \tag{1}$$

where W is a matrix of dimensions output_dim$\times$input_dim, and b a vector of size output_dim.

Another important element of hidden layers is the activation function. It introduces non-linearity into the output of the neuron and enables the network to model complex non-linear relationships in the input data. Probably the most popular activation functions are Sigmoid, ReLU and Tanh, given by the following equations and plotted in figure 2

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

$$\text{ReLU}(x) = \max(0, x) \tag{3}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4}$$

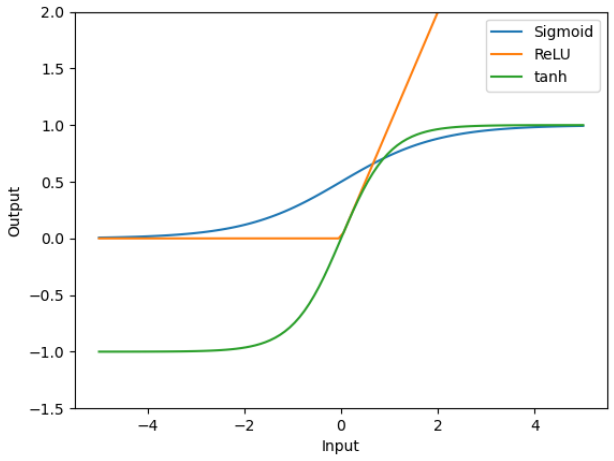

Fig. 2. Graphs of the most popular activation functions in neural networks

II. Deep Neural Network Implementation

A. Computational Nodes

Before proceeding into the explanation of the implementation of the DNN, we first need to understand the building

blocks of the network, computational nodes, the Tensor and the Parameter.

1) Tensor: In this lab assignment, we define a custom Tensor class, which is used to implement automatic differentiation for a simple neural network library. It represents a tensor in the computation graph, and stores its data, its gradient, the boolean flag, which represents whether it requires gradient calculation or not, its derivative function, and a list of the tensors that it depends on in the graph.

In the class they are stored in a data structure, which contains the following attributes: data, require grad, gradient, d, backptr. A **data** attribute stores a numpy array and a **require grad** attribute indicates whether this tensor requires gradient computation during backpropagation. If **require_grad** is set to **True**, then the tensor also has a **gradient** attribute that stores the gradient of the tensor with respect to the loss function, and a backward method to compute gradients using the chain rule. The backward pass is performed recursively on tensors in the **backptr** list that require gradient computation.

```
class Tensor:
    def __init__(self, data, require_grad=False):

        if isinstance(data, float):
            data = np.array([data,])
        if type(data) != np.ndarray:
            raise RuntimeError
            ("Input should be a numpy array")

        self.data = data
        self.require_grad = require_grad

        self.gradient = None
        self.d = None
        self.backptr = None
```

The **accumulate_gradient** method accumulates gradients from other tensors for this tensor during backpropagation. The **zero_grad** method sets the gradient of the tensor to zero, which is useful for clearing the gradients before performing a backward pass on a new batch of data.

The **backward** method performs the backward pass of the autograd algorithm. If a gradient tensor **g** is provided, it sets the gradient of this tensor to **g**. Then, if this tensor has a derivative function which is stored in **self.d**, it calls this function with the appropriate arguments. Finally, it recursively calls the backward method of all tensors in the **backptr** list that require gradient calculation and are of type Tensor.

```
    def backward(self, g=None):

        if self.d is not None:
            self.d(self.backptr, self.gradient)

        if self.backptr is not None:
            for o in self.backptr:
                if isinstance(o, Tensor)
                and not isinstance(o, Parameter):
                    if o.require_grad:
                        o.backward()
```

This custom Tensor class enables automatic differentiation for a simple neural network library, and is similar to the Tensor class used in popular deep learning frameworks like PyTorch and TensorFlow. However, this implementation is minimal and lacks some of the advanced features provided by modern deep learning frameworks, such as tensor broadcasting, slicing, advanced indexing. For example, PyTorch's **Tensor** also uses numpy arrays as its underlying data structure, but it also provides additional support for GPU tensors, which can significantly speed up computations on large models. PyTorch allows to move tensors between CPU and GPU using the .to() method, and perform computations on the GPU using the .cuda() method.

2) Parameter: The Parameter class inherits from the Tensor class, but it overrides the backward method to prevent backpropagation from a parameter node. This is because parameter nodes represent the learnable parameters of the network, and their gradients are computed during the backward pass and used to update the parameter values through optimization methods (e.g., SGD, Adam, etc.). Thus, it doesn't make sense to call their backward method, since they don't have any preceding computation (they didn't come from any operations on other tensors).

B. Deep Neural Network

To implement DNN, we were provided with the **Module** class, which contains an **__init__** method that takes no arguments, and a **parameters** method that returns a list of all the Parameter objects contained in the module. The **ModuleList** class is a subclass of list, used to store a list of Parameter or Module objects. Both **Module** and **ModuleList** classes have a **__call__** method, which simply calls the forward method of the module.

To define deep neural network with fully connected layers, we initialize the parameters of the network such as the number of input dimensions, output dimensions, hidden dimensions, and number of layers in the **__init__** method. We define the boolean parameter **tanh** that specifies whether to use the hyperbolic tangent activation function or not.

The **init_parameters** method initializes the weights and biases of the network. **Glorot_init** [1] is a function that initializes the weights of each layer according to a uniform distribution with range

$$-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}$$

(m and n being the dimensions of the weight matrix).

```
    def forward(self, x):
        x = affine_transform
        (self.input_proj, self.input_bias, x)
        x = relu(x)
        for W, b in zip(self.W, self.b):
            x = affine_transform(W, b, x)
```

```
            x = relu(x)
        x = affine_transform
        (self.output_proj, self.output_bias, x)
        return x
```

The forward pass of the network is defined in the forward method. It takes an input x, applies an affine transformation using the input projection matrix and input bias term, and then applies the ReLU activation function. Then, for each layer of the network, it applies an affine transformation using the weight matrix and bias term of that layer, and again applies the ReLU activation function. Finally, it applies an affine transformation using the output projection matrix and output bias term and returns the output of the network.

The **give_weight_matrix** and **give_bias** methods are static methods that create weight and bias matrices as Parameter objects, which are trainable parameters of the network. They take as input the dimensions of the weight and bias matrices and the **index n** to give a unique name to each weight and bias parameter.

C. Linear Network

We define a linear layer with input dimension **dim input** and output dimension **dim output**. The weight matrix W and bias vector b are initialized in the constructor using the **glorot_init** and **zero_init** functions respectively.

The forward function computes the output of the linear layer using the **affine_transform** function that calculates the dot product of W and x, adds the bias term b, and returns the output. The **LinearNetwork** class is often used as a building block in more complex neural networks, where it is combined with non-linear activation functions to create non-linear mappings between the input and output [2].

Our implementation is similar to the PyTorch's fully connected layer **nn.Linear** which also performs linear transformation of the input data. However, PyTorch implementation has many additional features and options that can be used to customize the behavior of the layer. For example, nn.Linear allows to set different weight and bias initialization schemes, specify activation functions, and add dropout or batch normalization layers.

III. Gradients retrieval

In this section we provide detailed mathematical explanation on how we computed the gradients, and the affine transforms which we used for building the Deep Neural Network.

A. ReLU

The REctified Linear Unit is one of the most popular activation functions, which is based of the idea of neither wanting negative activations nor forcing activations between a range. It is piecewise defined as $ReLU(x) = \max(0, x)$ and its piecewise derivative is:

$$\frac{d}{dx}ReLU(x) = \begin{cases} \frac{dx}{dx} & x > 0 \\ \frac{d0}{dx} & otherwise \end{cases} = \begin{cases} 1 & x > 0 \\ 0 & otherwise \end{cases}$$

Note that, as a solution to the fact that $\frac{d}{dx}ReLU(x)$ does not exist when $x = 0$, we set it arbitrarily to value 0.

B. Tanh

Tanh is another popular activation function, and while it has a similar shape to sigmoid, it has a different range, and a different gradient. Tanh is defined as:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

. Therefore, we can retrieve its gradient as:

$$\frac{d}{dx}\tanh(x) = \frac{\cosh(x)\frac{d\sinh(x)}{dx} - \sinh(x)\frac{d\cosh(x)}{dx}}{\cosh(x)^2} =$$

$$\frac{\cosh(x)^2 - \sinh(x)^2}{\cosh(x)^2} = 1 - \left(\frac{\sinh(x)}{\cosh(x)}\right)^2 = 1 - \tanh(x)^2$$

C. Affine function

In the case of the affine function, defined as $y = f(x, W, b) = Wx + b$, we need to compute three different gradients: with respect to W, b and x. Notice that because y is a vector, the concept of gradient generalizes to a differential, which will be a multidimensional tensor depending on the variable we use to derive:

$$\begin{aligned} \frac{\partial y_k}{\partial W_{ij}} &= x_j \delta_{ki} \\ \frac{\partial y_k}{\partial b_i} &= 1 \\ \frac{\partial y_k}{\partial x_i} &= W_{ki} \end{aligned} \tag{5}$$

D. Negative Log Likelihood Loss

Let's first consider that the Negative Log Likelihood Loss (NLL loss) is defined as:

$$NLL(x, gold) = -\log\left(\frac{\exp(x_{gold})}{\sum_j \exp(x_j)}\right) =$$

$$= -x_{gold} + \log\left(\sum_{j=1}^{D}\exp(x_j)\right)$$

where x is a single datapoint of dimension D and $gold$ is the index of the correct class of the datapoint x. Therefore, in order to retrieve the gradient over x, we first find that:

$$\frac{\partial}{\partial x_i} - x_{gold} + log\left(\sum_{j=1}^{D}\exp(x_j)\right) = -\mathbb{1}(gold = i) + \frac{e^{x_i}}{\sum_{j=1}^{D} e_j^x}$$

where $\mathbb{1}(gold = i) = \begin{cases} 1 & gold = i \\ 0 & otherwise \end{cases}$

Finally, we retrieve the gradient:

$$\frac{d}{dx} - x_{gold} + log\left(\sum_{j=1}^{D}\exp(x_j)\right) = -\text{one\_hot(gold)} + \text{softmax}(x)$$

where $one\_hot(gold)$ is a vector of zeros of dimension D with only one value 1 in the position corresponding to the index defined by $gold$ (e.g. $gold = 3, D = 5, one\_hot(gold) = [0, 0, 0, 1, 0]$).

E. Chain rule

1) Activation functions: Suppose we are given a vector input z, and we apply an activation function. This activation function will consist of the same one variable function f applied independently to each component of z, so that $y_i = f(z_i)$. Now assume that we are given the gradient of the loss function with respect to the output (i.e. $\partial\mathcal{L}/\partial y_i$), the way to get the gradient with respect to the input of the activation function is to perform an element-wise multiplication, this time with the derivative of f applied to every component of z:

$$\frac{\partial \mathcal{L}}{\partial z_i} = \frac{\partial \mathcal{L}}{\partial y_i} * f'(z_i) \tag{6}$$

2) Affine function: Similarly, let's say we are given the gradient of the loss with respect to the output of the affine transformation, and we wish to compute the gradients with respect to the inputs W, x, b. Using the differentials in equation (5), we can derive the following gradients:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y} x^T,$$
$$\frac{\partial \mathcal{L}}{\partial x} = W^T \frac{\partial \mathcal{L}}{\partial y},$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial y}$$

We implement these rules in the backward methods of the different activation functions and linear layers. For example, this is a simplified version of the **backward_affine_transform** function

```
def backward_affine_transform(backptr, g):
    W, b, x = backptr
    if W.require_grad:
        W.accumulate_gradient(g@x.data.T)

    if b.require_grad:
        b.accumulate_gradient(g)

    if x.require_grad:
        x.accumulate_gradient(W.data.T@g)
```

IV. Experiments

In this experiment, we investigate the effect of different learning rates as well as the momentum values on the training of a deep neural network with 2 layers and 100 hidden size. We also tried to experiment with 6 layers and hidden sizes of 200, 150, 100, 50, 25, and 10. We used the MNIST dataset for our experiments, which consists of 40,000 training images and 10,000 tests & validation images belonging to 10 different classes.

Using the implemented neural network from scratch, we trained it using the momentum stochastic gradient descent optimizer. We trained the network for 20 epochs and recorded the training and test accuracies and their losses for each combination of learning rate and momentum values.

The following figure shows the training and test accuracies for 2 layers with 100 hidden size and different combinations of learning rates and momentum values:

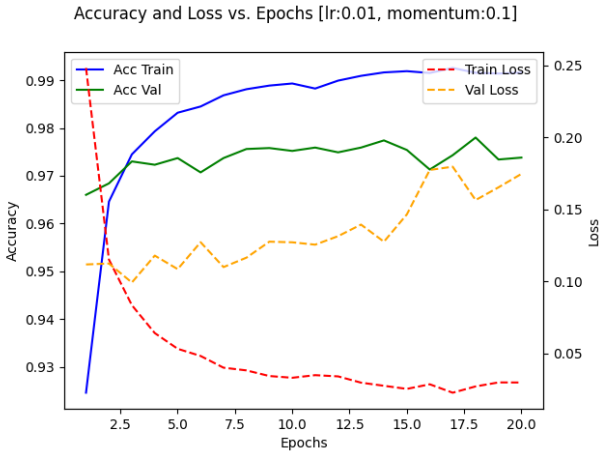

Fig. 3. Result with 0.01 learning rate and 0.1 momentum

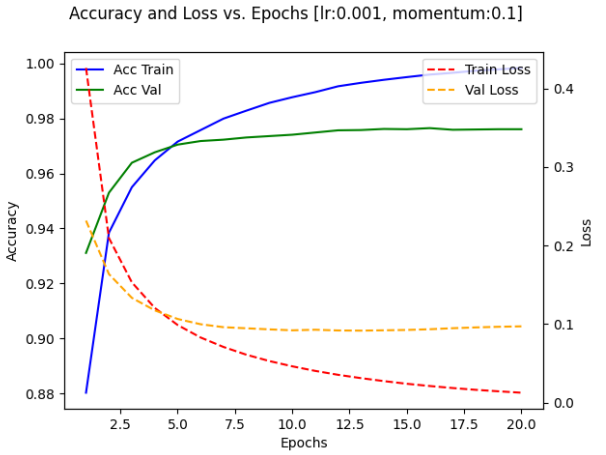

Fig. 4. Result with 0.001 learning rate and 0.1 momentum

From the figure, we observe that the network achieves high training and validation accuracies this is because the dataset is relatively small with only 10 classes containing grayscale images of size 28x28 pixels. Moreover, the images in MNIST are already pre-processed and normalized, making it easier for neural networks to learn the patterns and features in the data.

In Figure 3 and 5 we analyze that the graph is overfitting. When the learning rate is too high, the optimizer can overshoot the optimal parameters and lead to unstable training. We believe that a learning rate of 0.001 is considered to be a reasonable starting point for training neural networks (Figure 4 and 6). This could also be improved with a learning scheduler such as cosine annealing, polynomial decay, etc.

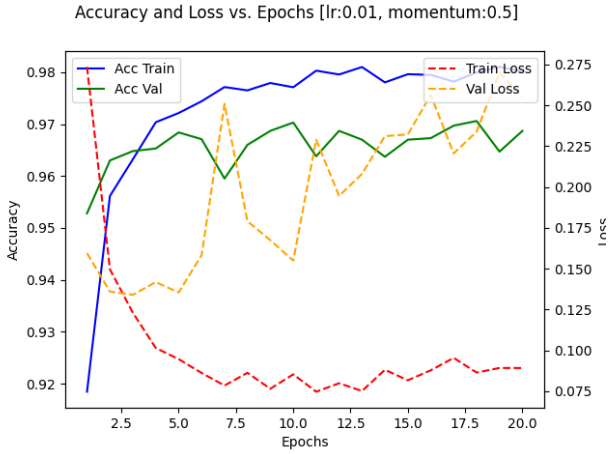

Fig. 5. Result with 0.01 learning rate and 0.5 momentum

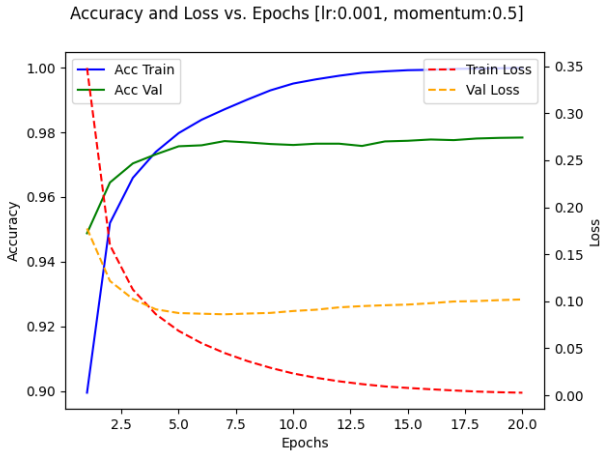

Fig. 6. Result with 0.001 learning rate and 0.5 momentum

We have noticed that increasing the momentum value can enhance the performance of the neural network, particularly for lower learning rates. However, it is crucial to be mindful of setting the momentum value too high in an optimizer, as it can cause the model to memorize the training data too quickly and lead to overfitting, Figure 8. Moreover, when a high learning rate and high momentum are combined (Figure 7), the training process can become unstable, with the weights and bias values changing rapidly, making it challenging for the model to converge to an optimal solution.

Inspired by the paper [3] and after conducting various experiments, we identified the optimal hyperparameters for our model. We attempted to add more layers to the model but were limited by computational constraints, which forced us to reduce the size of the model by using more reasonable values. We settled on a model architecture consisting of six layers with sizes of 200, 150, 100, 50, 25, and 10. Through our experimentation, we achieved a validation accuracy of 97.99%, Figure 9.

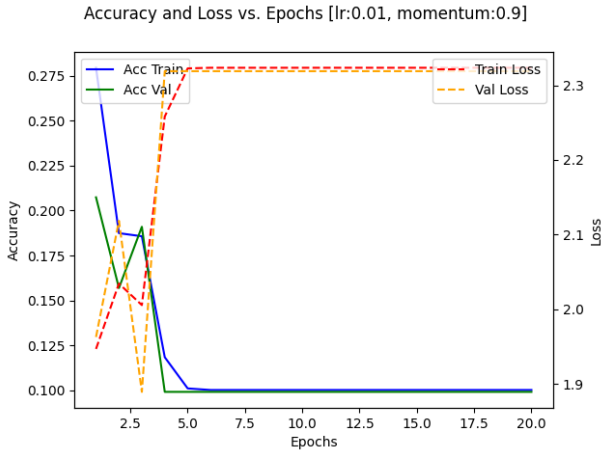

Fig. 7. Result with 0.01 learning rate and 0.9 momentum

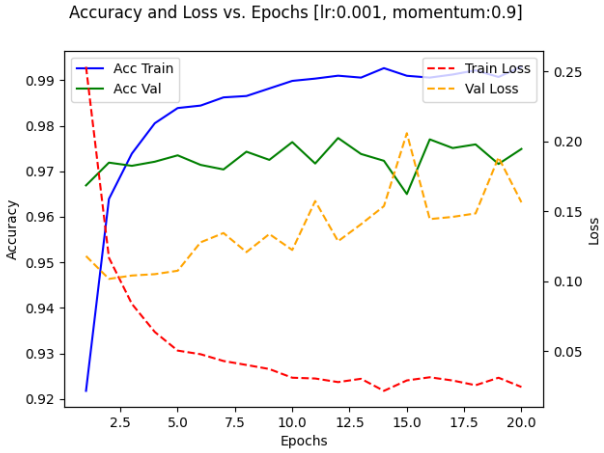

Fig. 8. Result with 0.001 learning rate and 0.9 momentum

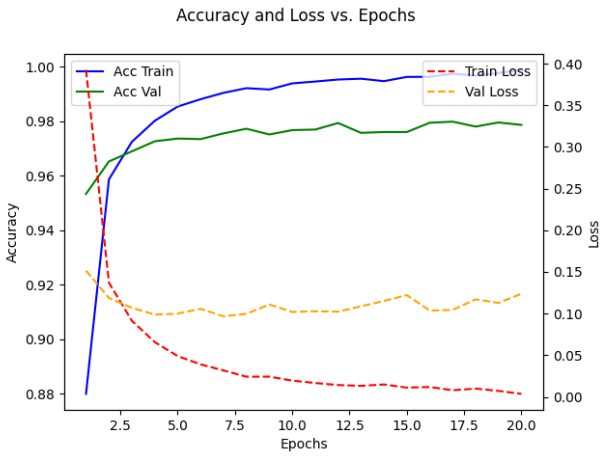

Fig. 9. Result with 0.001 learning rate and 0.3 momentum

V. Conclusion

In conclusion, this project involved the implementation of a simple auto-diff library that mimics Pytorch's autograd mechanism. This mechanism is a fundamental component of deep learning, allowing for automatic computation of gradients, which are used to update the model's parameters and minimize the loss function. While our implementation may not be as comprehensive as Pytorch's, we believe it has been a valuable learning experience as it has allowed us to gain a deeper understanding of how auto-diff works and of Pytorch's underlying principles.

Based on our experiments, it is important to find a balance between the learning rate and momentum. It is generally recommended to start with a low learning rate and momentum and gradually increase them while monitoring the training process. This can help to prevent overshooting, unstable training, and overfitting, while also allowing for fine-tuning of the model.

References

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," pp. 249–256, 2010.

[2] S.-Y. R. Li, Q. T. Sun, and Z. Shao, "Linear network coding: Theory and algorithms," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 372–387, 2011.

[3] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep big simple neural nets excel on handwritten digit recognition," *Neural Computation*, vol. 22, no. 12, pp. 4–5, 2010.