Tensor Computing in Neural Networks

Introduction

Tensor computing is fundamental to neural networks, as it enables efficient representation and manipulation of multidimensional data. Neural networks process inputs, weights, and activations in the form of tensors, allowing for structured mathematical operations that facilitate learning and inference.

Representation of Data as Tensors

In neural networks, data is structured as tensors—multidimensional arrays that generalize vectors and matrices. The dimensionality of these tensors depends on the type of data being processed:

- **Scalars (0D tensors):** Single numerical values, often used for loss values.
- **Vectors (1D tensors):** Arrays of numbers, commonly representing input features or output activations.
- **Matrices (2D tensors):** Used to store batches of data, weight matrices in fully connected layers, or transformation matrices.
- **Higher-Dimensional Tensors (3D, 4D, etc.):** Essential for handling complex data like images (height × width × channels) or sequences (timesteps × features).

Tensor Operations in Neural Networks

Neural networks rely on tensor operations to perform computations efficiently. Some key operations include:

- **Element-wise operations:** Addition, multiplication, and activation functions (e.g., ReLU, sigmoid) applied independently to each element of a tensor.
- **Matrix multiplications:** Used in dense (fully connected) layers, where input tensors are multiplied by weight tensors.
- **Tensor contractions:** More general forms of matrix multiplication, seen in attention mechanisms and certain types of convolutions.
- **Convolutions:** Used in convolutional neural networks (CNNs) to extract spatial features from input tensors.
- **Reductions:** Summation, averaging, and other operations that aggregate tensor values, often used in pooling layers and loss calculations.
- **Broadcasting:** Allows operations between tensors of different shapes by automatically expanding dimensions where necessary.

Tensor Computation for Forward and Backward Passes

Neural network training consists of two key phases: forward propagation and backpropagation. Both rely heavily on tensor computations.

- **Forward propagation:** Computes the network's output by applying a sequence of tensor operations, transforming input tensors through layers.
- **Backpropagation:** Computes gradients using tensor differentiation. It relies on the chain rule to propagate errors backward, updating weights via tensor-based gradient descent.

Defining Neural Networks Explicitly

In the following sections, we explain the main computations that occur in fully-connected neural networks. We start with the most basic neural network (a perceptron) and then move on to a network with a single hidden layer.

We use the notation w.r.t. to mean "with regard to".

In each case, we explain:

- The forward pass: computing the output from the input
- The backward pass: coputing the gradients of the loss function w.r.t. the weights
- Training: updating the weights using basic gradient descent
-

A Single Perceptron

For a single perceptron, the functionality is presented in Figure 1.

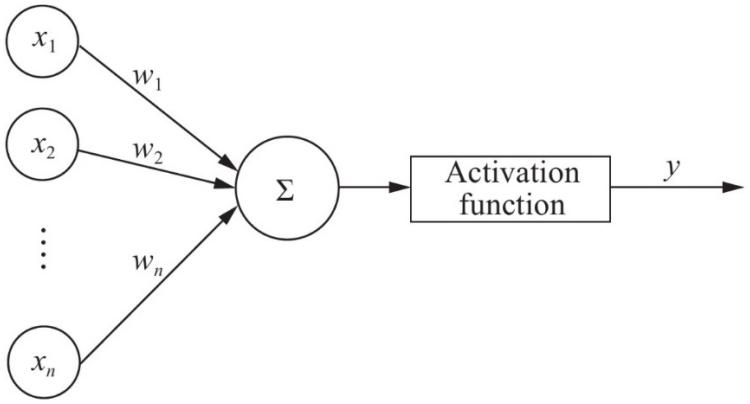

Figure 1. A perceptron that receives inputs X and computes output y using an activation function.

In the following paragraphs, we assume that the activation is the sigmoid function:

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

The perceptron receives inputs $X = [x_1, x_2, \dots, x_n]$, where n is the number of features (i.e. the number of data characteristics). The inactivated output of the perceptron is:

$$z = \sum w_i x_i + b$$

Where w_i are the weights corresponding to each input x_i, and b is the bias (a free terms that is required so the perceptron generalizes better). The percetron applies the activation function, resulting in its final output:

$$\hat{y} = \sigma(z) = \sigma\left(\sum w_i x_i + b\right)$$

Obtaining the output from input data is referred to as the **forward pass**.

We use the notation $\hat{y}$ ("y hat") to sinnlify the actual output of the perceptron. The regular notation y (y without the hat) is used to signify the desired output (from the training data).

Training the perceptron involves adjusting weights w_i and bias b so as to minimize the difference between actual outputs $\hat{y}$ and desired outputs y. We compute this difference using a **loss function**. We use the following loss function:

$$L = \frac{1}{2}(\hat{y}_i - y_i)^2$$

Where $\hat{y}_i$ and y_i are the actual and desired outputs for input X_i, i = [1..m], where m is the number of instances in the training data set.

Updating the parameters w_i and b requires that we compute the gradient of the loss function w.r.t. the parameters. We need to compute:

$$\frac{\partial L}{\partial w_i} \text{ and } \frac{\partial L}{\partial b}$$

In the general case, computing the gradient is carried out from the output of the neural network (where we determine the loss function) toward the input. We compute the gradients through a backward pass through the network, and the entire process is referred to as **backpropagation**. Computing the gradients is one of the most computationally-demanding aspects of training a neural network and one of the main reasons why training deep networks takes so long.

A formula that is fundamental to computing gradients is the chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

The gradient of the loss function L w.r.t. activations $\hat{y}$ is:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$$

The gradient of the activations w.r.t. the inactivated output z is (the derivative of the sigmoid function):

$$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y})$$

Considering that $z = \Sigma\, w_i\, x_i + b$, the gradient of inactivated output z w.r.t weights w_i is

$$\frac{\partial z}{\partial w_i} = x_i$$

And the gradient of inactivated output z w.r.t the bias is

$$\frac{\partial z}{\partial b} = 1$$

Using the chain rule, we decompose the gradient of the loss w.r.t weights w_i as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

Substituting the previous results gives us:

$$\frac{\partial L}{\partial w_i} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y}) \cdot x_i$$

Similarly, for the bias:

$$\frac{\partial L}{\partial b} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$$

Once the gradients of the loss w.r.t. the network parameters are determined, we update the weights using an optimization algorithm, such as **gradient descent**:

$$w_i = w_i - \eta \frac{\partial L}{\partial w_i}$$

$$b = b - \eta \frac{\partial L}{\partial b}$$

Where η (“eta”) is the learning rate.

The Case of a Neural Network

We present a neural network with a single hidden layer in Figure 2. Its functionality is similar to the previosuly-discussed perceptron.

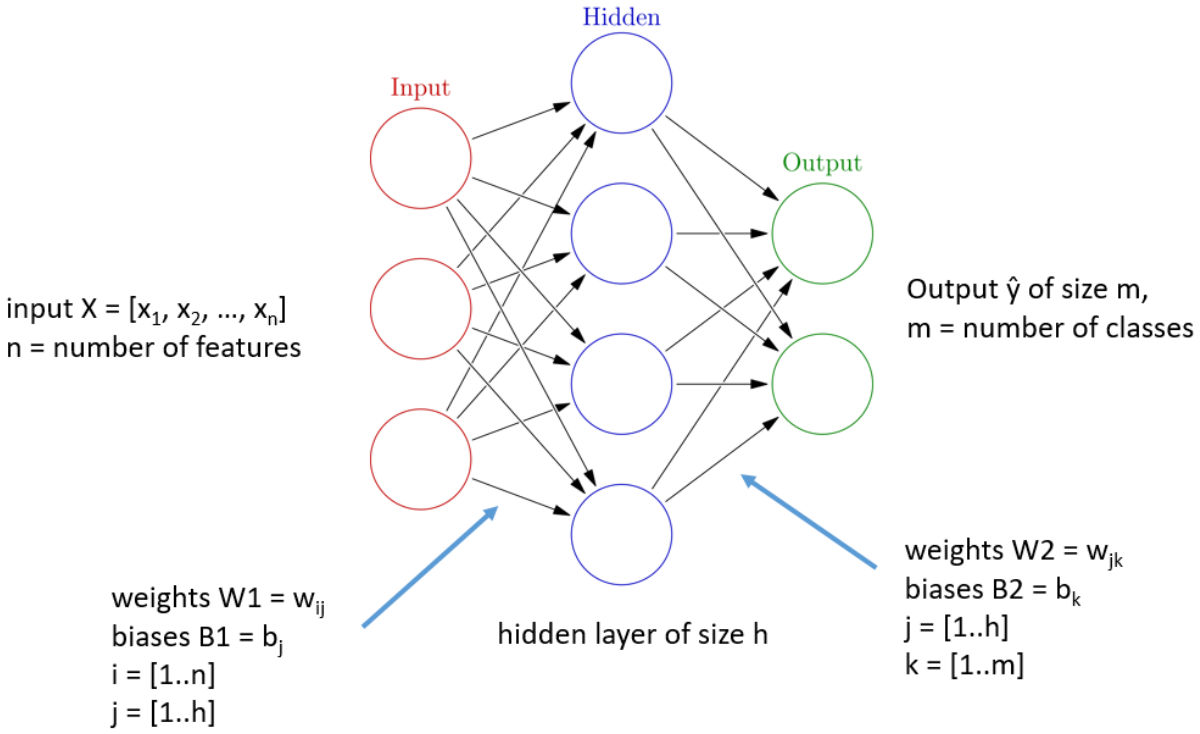

Figure 2. A neural network with a single hidden layer.

X is the input vector containing the features (characteristics of the data)

W_1 is a weight matrix containing weights inbetween the input layer and the hidden layer. B_1 is the corresponding bias vector.

W_2 is the weight matrix inbetween the hidden layer and the output layer. B_2 is the correpsonding bias vector.

Output $\hat{y}$ is the actual output of the network. For a classification problem, it is a distribution over the set of classes (it contains a probability for each possible class).

Forward propagation

The objective of forward propagation is to compute outputs $\hat{y}$ from inputs X.

First, we compute the values of the hidden layer:

$$z_1 = W_1 X + B_1$$
$$a_1 = \sigma(z_1)$$

z_1 are the inactivated values of the hidden layer. We apply activation function σ (we also assume sigmoid activation), resulting in hidden layer activations a_1.

We then compute the values of the output layer:

$$z_2 = W_2 X + B_2$$
$$\hat{y} = soft\max(z_2)$$

z_2 are the inactivated values of the output layer (also referred to as *logits*). We apply the *softmax* activation function to convert these into a probability distribution (i.e. to tranform a vector of real values into a vector of probabilities)

$$soft\max(x_i) = \frac{e^{x_i}}{\sum e^{x_i}}$$

Backpropagation

As in the case of the perceptron, training the neural network involves ajusting weights W and biases B so as to minimize the difference between network outputs $\hat{y}$ and y. Assuming a multiclass clasification problem (more than 2 classes), we measure this difference using Cross Entropy Loss:

$$L = -\sum y_i \log(\hat{y}_i)$$

Backpropagation involves computing the gradients of the loss w.r.t the weights and biases.

That means we have to compute:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial w_{ij}}$$

$$\frac{\partial L}{\partial B_1} = \frac{\partial L}{\partial b_j}$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial w_{jk}}$$

$$\frac{\partial L}{\partial B_2} = \frac{\partial L}{\partial b_k}$$

i = 1..n, n = number of features (size of the input layer)

j = 1..h, h = size of the hidden layer

k = 1..m, m = number of classes (size of the output layer)

We start with the gradients corresponding to the output layer, where we have direct access to the loss function. We then work backwards towards the hidden layer, where the required computations will depend on the results obtained in the output layer. This way, gradient computation propagates backwards through the neural network, starting from the output layer and ending at the input layer.

Using the chain rule, we decompose the gradient of the loss w.r.t. the weights in the output layer as follows:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2}$$

And

$$\frac{\partial L}{\partial B_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial B_2}$$

Considering the expression of the loss function and the fact that we apply softmax in the output layer, the gradient of the loss w.r.t. the output layer values (logits) is:

$$\frac{\partial L}{\partial z_2} = \hat{y} - y$$

Considering that $z_2 = W_2\, a_1 + B_2$, the derivative of z_2 w.r.t to W_2 is just a_1:

$$\frac{\partial z_2}{\partial W_2} = a_1$$

By substituting in the chain rule, we get:

$$\frac{\partial L}{\partial W_2} = (\hat{y} - y) a_1$$

Considering that $z_2 = W_2\, a_1 + B_2$, the derivative of z_2 w.r.t to B_2 is 1:

$$\frac{\partial z_2}{\partial B_2} = 1$$

By substituting in the chain rule, we get:

$$\frac{\partial L}{\partial B_2} = \hat{y} - y$$

In the hidden layer, each hidden neuron contributes to multiple output neurons. Therefore, the total hidden error is the sum of the weighted contributions from all output neurons:

$$\frac{\partial L}{\partial z_1} = W_2 \frac{\partial L}{\partial z_2}$$

Since the hidden layer also applies activation function $\sigma(z_1)$, we adjust its effect using its derivative:

$$\frac{\partial L}{\partial z_1} = W_2 \frac{\partial L}{\partial z_2} \sigma^{'}\left(z_1\right)$$

For sigmoid activation, $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

For the gradient of the loss w.r.t. the weights W_1 from input to the hidden layer, we apply the chain rule as follows:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1}$$

$$\frac{\partial L}{\partial B_1} = \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial B_1}$$

Since $z_1 = W_1 X + B_1$, its derivative w.r.t. W_1 is just input X:

$$\frac{\partial z_1}{\partial W_1} = X$$

And its derivative w.r.t. biases B_1 is 1:

$$\frac{\partial z_1}{\partial B_1} = 1$$

By substituting in the chain rule, we get:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} X$$

$$\frac{\partial L}{\partial B_1} = \frac{\partial L}{\partial z_1}$$

Therefore, we compute the gradients of the loss w.r.t. the parameters of the input-hidden layer using the results previously-determined from the output layer:

$$\frac{\partial L}{\partial W_1} = W_2 \frac{\partial L}{\partial z2} \sigma^{'}\left(z_1\right) X$$

$$\frac{\partial L}{\partial B_1} = W_2 \frac{\partial L}{\partial z2} \sigma^{'}\left(z_1\right)$$

In order to update the weights and biases, we use a gradient descent-type aproach, similar to the case of the perceptron:

$$W_1 = W_1 - \eta \frac{\partial L}{\partial W_1}$$

$$B_1 = B_1 - \eta \frac{\partial L}{\partial B_1}$$

$$W_2 = W_2 - \eta \frac{\partial L}{\partial W_2}$$

$$B_2 = B_2 - \eta \frac{\partial L}{\partial B_2}$$

Tasks

1. Implement a perceptron (a single neuron) with sigmoid activation, explicitly, using only PyTorch tensors and tensor operations (no PyTorch Linear module, no automatic backpropagation, no ready-made optimizers).
   1.1. Implement a perceptron using the classic Pytorch approach, and compare its performance and convergence with the explicitly-developed one.

2. Implement a neural network with a single hidden layer with sigmoid activation, explicitly, using only PyTorch tensors and tensor operations (no PyTorch Linear module, no automatic backpropagation, no realy-made optimizers).
   2.1. Implement the same network using the classic PyTorch approach, and compare its performance and convergence with the explicitly-developed one.