Vectorization

This handout follows a notation where individual matrices and vectors are shown in bold, where matrices are capital and vectors are not. However, when talking about a particular single element of a vector/matrix lowercase non-bold letter is used. Another notational nuance is that indices of matrices or vectors are written in subscript and n^th element from a collection is represented using a superscript. As an example x_j^i is the j^th element of the matrix $\mathbf{x}^i$ (we have more than one vector $\mathbf{x}$, and each $\mathbf{x}$ has some number of elements). All superscripts represent this, unless otherwise stated (such as for power).

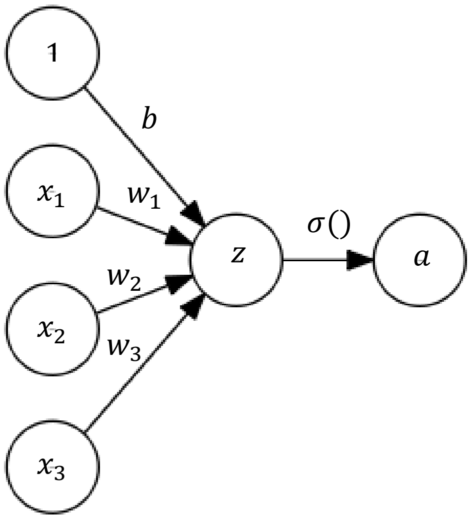
Linear Model as a Perceptron

In class we discussed linear regression as a very simple machine learning model which fits a linear polynomial function to the data by estimating the W coefficients of the equation:

$$\sum_{i=1}^{n} w_i x_i + b$$

The coefficients are sometimes called the parameters or weights of the model. The term b, is called the bias and acts as the intercept or offset for the learned linear model.

We also discussed a more visual and mechanical interpretation of a linear model; the perceptron (shown on the right). This graphical model implements the exact same model as the linear regression equation above. The example on the right shows logistic regression because the sigmoid function is applied to the weighted sum. The $\boldsymbol{w}$ vector replaces the parameters as weights.



$$z = \sum_{i=1}^{n} w_i x_i + b$$

$$a = \sigma(z)$$

Some textbooks use an even more concise representation for Z, both for notational and computational reasons.

$$z = \sum_{i=0}^{n} \overline{w_i}\overline{x_i}$$

The new $\overline{\boldsymbol{w}}$ and $\overline{\boldsymbol{x}}$ vectors have one extra element in each (the 0th element). $\boldsymbol{w}$ has been augmented with b and $\boldsymbol{x}$ now has a 1. This is mathematically identical to the equation above but uses just a single multiplication operation instead of the extra addition. Shortcuts like this are how all machine learning libraries speed up training. We will, however, not be using this formulation.

Vectorized form of Perceptron

One advantage of using a perceptron as a linear model is that it is very simple to convert this structure into a multi layered version and creating a non-linear model. The original name of Neural Networks was Multi-Layered Perceptrons for this exact reason. But first, we need a more compact way to represent the mathematical equation, for that we will use linear algebra.

$$z = \boldsymbol{xw} + b$$

This equation for z looks eerily similar to the previous one, so let's take a closer look. The vector $\boldsymbol{x}$ has shape 1×3 (for the example in the diagram) and $\boldsymbol{w}$ has 3×1, b is just a single number but it can be thought of as a 1×1 vector/matrix.

We can see, using the matrix multiplication rules that the output of this equation will be a single number, identical to the previous versions. We can also use this matrix notation to describe the learning procedure for the perceptron with least squares error. The weight update step discussed in the class is:

$$w_i = w_i - \eta \frac{\delta E}{\delta w_i}$$

$$E = \sum_{\forall i} (t_i - y_i)^2$$

The superscript here represent power

Where $\boldsymbol{y}$ is the model output or $\boldsymbol{z}$ which depends on $\boldsymbol{w}$, and η is the learning rate (step size). We want to minimize E so we use its derivative with respect to $\boldsymbol{w}$. We can vectorize this step very easily as well since a perceptron has a single output, so E will be a single number. We can obtain a vector containing all the partial derivatives:

$$\Delta \boldsymbol{w} = \left[{}^{\delta E}/_{\delta w_1} \quad {}^{\delta E}/_{\delta w_2} \quad \cdots \quad {}^{\delta E}/_{\delta w_n} \right]$$

Now the weight update step simply becomes:

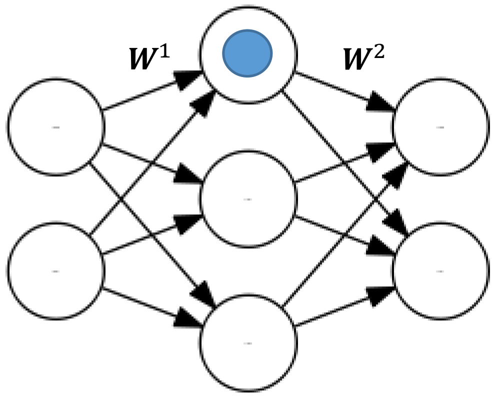
$$\boldsymbol{w} = \boldsymbol{w} - \eta \, \Delta \boldsymbol{w}$$

Not only is the vectorized form more concise, it is slightly more easy to understand as well because you do not need to keep track of indices of matrices.

Vectorized Multi-Layer Perceptron

Now we will join a few perceptrons together to form a Neural Network. We can easily accomplish this by passing on the output of the first layer/set of perceptrons as the input to every successive layer.

First, the boring stuff… nomenclature. The model on the right is sometimes called a 2-layer model because it has 2 sets of trainable perceptrons/weight matrices. In some other texts (and the naming we will follow) it is called a 3-layer model because visually it has 3 sets of 'nodes' (the first set is the input layer which is technically constant). Be careful about this difference when implementing because with our naming scheme, the 'first' is not trainable because it has no weights.



The step by step derivation and explanation of these equations can be found in the lecture slides along with a worked example. This handout will just list the equations and describe the notation a bit. The equations correspond to the same 3-layer network shown above with sigmoid activation on both layers and MSE loss.

Now let's see how this structure is built. The first perceptron in the second layer (marked with a blue circle) has 2 inputs and therefore 2 weights. Previously we described this with a 2×1 vector, but now we have 3 such neurons/perceptrons. Let's stack these 3 weight vectors next to each other as columns of $\boldsymbol{W}^1$ which is a 2×3 matrix. Each perceptron outputs a single number, so a layer of 3 perceptrons will output a vector ($\boldsymbol{z}$) of shape 1×3. Each neuron will also have a separate bias (not shown on the diagram) so it will also be a vector of shape 3×1. The addition still works out because of 'broadcasting' in a Python implementation, but mathematically you can simply transpose $\boldsymbol{b}$.

Forward Pass

$$\boldsymbol{z}^1 = \boldsymbol{x}\boldsymbol{W}^1 + \boldsymbol{b}^1$$

$$\boldsymbol{a}^1 = \sigma(\boldsymbol{z}^1)$$

This becomes the input to the next layer of neurons/perceptrons with their own weight matrix constructed in a similar fashion.

$$\boldsymbol{z}^2 = \boldsymbol{a}^1\boldsymbol{W}^2 + \boldsymbol{b}^2$$

$$\boldsymbol{a}^2 = \sigma(\boldsymbol{z}^2)$$

This is the entire forward pass of a neural network in [basically] 2 matrix products.

Backward Pass

$$\boldsymbol{\phi}^2 = \boldsymbol{a}^2 - \boldsymbol{y}$$

$$\boldsymbol{deriv}^2 = \boldsymbol{a}^2(1 - \boldsymbol{a}^2)$$

$$\boldsymbol{\Delta w}^2 = \boldsymbol{\phi}^2 \odot \boldsymbol{deriv^2}$$

$$\boldsymbol{\phi}^1 = (\boldsymbol{W}^2(\Delta\boldsymbol{W}^2)^T)^T$$

$$\boldsymbol{deriv}^1 = \boldsymbol{a}^1(1 - \boldsymbol{a}^1)$$

$$\boldsymbol{\Delta w}^1 = \boldsymbol{\phi}^1 \odot \boldsymbol{deriv}^1$$

The $\boldsymbol{\phi}$ term is derivative of error which can directly be obtained for the output layer as we have the ground truth data $\boldsymbol{y}$. This is not possible for the intermediate/hidden layers, which is where the chain rule and backpropagation algorithm comes in. We can calculate the $\boldsymbol{\phi}$ of any hidden layer using the $\boldsymbol{\Delta w}$ of the next layer, almost as if the error is *propagating backwards*. The $deriv$ terms are the derivatives of the activation function with the particular value of $\boldsymbol{z}$. This term changes if you change the activation function of a layer. The $\odot$ operator is called the Hadamard product and is a simple elementwise multiplication.

Technically, the $\boldsymbol{\Delta w}$s should essentially be used as is in the weight update equation (similar to the perceptron update equation shown previously). The way this handout describes the equations is that it splits the chain rule into two parts. The red portion is represented by $\boldsymbol{\Delta w}$ in the backward pass step, and the purple portion is calculated in the weight update equations.

$$\frac{\delta \boldsymbol{E}}{\delta \boldsymbol{a}} \cdot \frac{\delta \boldsymbol{a}}{\delta \boldsymbol{z}} \cdot \frac{\delta \boldsymbol{z}}{\delta \boldsymbol{W}}$$

There is no particular reason to setup the structure this way (it is again, an arbitrary choice). ***However, the automatic tests provided to you with Assignment 1 use this formulation.***

Weight Update

$$\boldsymbol{W}^2 = \boldsymbol{W}^2 - \eta\,((\boldsymbol{\Delta w}^2)^T \boldsymbol{a}^1)^T$$

$$\boldsymbol{b}^2 = \boldsymbol{b}^2 - \eta \sum_{\forall i} \boldsymbol{\Delta w}^2{}_i$$

$$\boldsymbol{W}^1 = \boldsymbol{W}^1 - \eta\,((\boldsymbol{\Delta w}^1)^T \boldsymbol{x})^T$$

$$\boldsymbol{b}^1 = \boldsymbol{b}^1 - \eta \sum_{\forall i} \boldsymbol{\Delta w}^1{}_i$$

Importance of Activation Functions

We have seen that the usage of an activation function is integral to the entire learning process of a neural network as it affects the full mathematical pipeline. But let's also build some intuition as to why it is so important.

Consider the following equation, which describes the feed forward step of a 3-layer neural network. You can suppose that the bias is either 0 or is incorporated in the matrices (like the concise form we discussed above).

$$\boldsymbol{y} = f(f(\boldsymbol{x}\boldsymbol{W}^1)\boldsymbol{W}^2)$$

The f is our activation function, here it is same for both layers but that is an arbitrary choice. To simulate the absence of the activation function let's use $f(x) = x$ as our linear activation function which essentially does nothing.

$$\boldsymbol{y} = \boldsymbol{x}\boldsymbol{W}^1\boldsymbol{W}^2$$

$$\boldsymbol{y} = \boldsymbol{x}(\boldsymbol{W}^1\boldsymbol{W}^2)$$

$$\boldsymbol{y} = \boldsymbol{x}\boldsymbol{W}^\alpha$$

We can see that using a linear activation function (same as not using an activation function) reduces the equation to a form which is pretty much the same as linear regression. That was a linear model which could not learn complex non-linear relationships in the data (limited to the usage of linear regression we have discussed).

So the non-linear powerful nature of a Neural Network depends completely on the presence of a non-linear activation function which is why these functions are sometimes referred to as 'non-linearities'.