Foundational Components of Neural Networks

Content is taken from the book Natural Language Processing with PyTorch by Brian McMahan, Delip Rao.

The perceptron:

Each perceptron unit has an input (x), an output (y), and three "knobs": a set of weights (w), a bias (b), and an activation function (f). The weights and the bias are learned from the data, and the activation function is handpicked depending on the network designer's intuition of the network and its target outputs. Mathematically, we can express this as follows:

y = f (wx + b)

It is usually the case that there is more than one input to the perceptron. We can represent this general case using vectors. That is, x, and w are vectors, and the product of w and x is replaced with a dot product:

y = f (wx + b)

The activation function, denoted here by f, is typically a nonlinear function. A linear function is one whose graph is a straight line. In this example, wx+b is a linear function. So, essentially, a perceptron is a composition of a linear and a nonlinear function. The linear expression wx+b is also known as an affine transform.

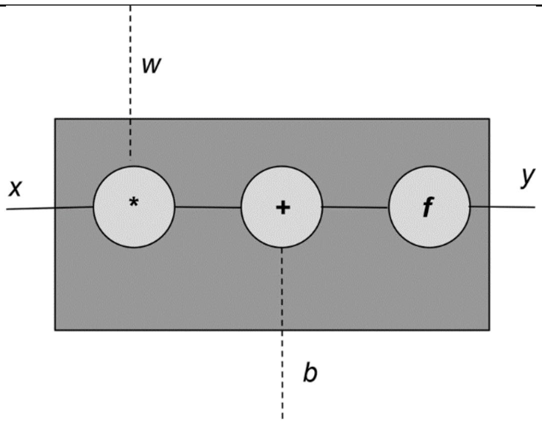
Code:

```
import torch
import torch.nn as nn

class Perceptron(nn.Module):
    """ A perceptron is one linear layer """
    def __init__(self, input_dim):
        """
        Args:
            input_dim (int): size of the input features
        """
        super(Perceptron, self).__init__()
        self.fc1 = nn.Linear(input_dim, 1)

    def forward(self, x_in):
        """The forward pass of the perceptron

        Args:
            x_in (torch.Tensor): an input data tensor
                x_in.shape should be (batch,
num_features)
        Returns:
            the resulting tensor. tensor.shape should be
(batch,).
        """
        return torch.sigmoid(self.fc1(x_in)).squeeze()
```

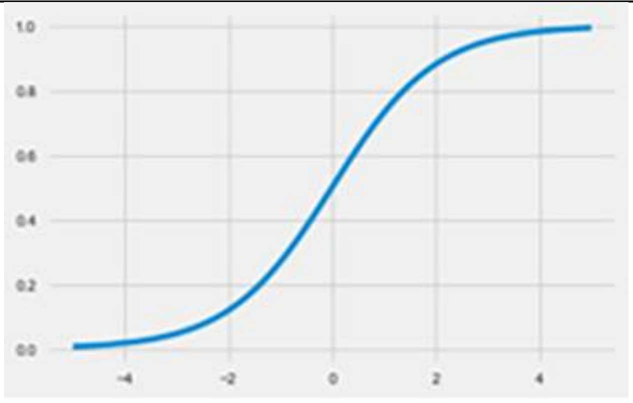


Activation Functions: Activation functions are nonlinearities introduced in a neural network to capture complex relationships in data.

a. **Sigmoid:** It takes any real value and squashes it into the range between 0 and 1. Mathematically, the sigmoid function is expressed as follows:
   $f(x)=1 / (1 + e^{-x})$
   It is easy to see from the expression that the sigmoid is a smooth, differentiable function. torch implements the sigmoid as torch.sigmoid()

```
import torch
import matplotlib.pyplot as plt

x = torch.range(-5., 5., 0.1)
y = torch.sigmoid(x)
plt.plot(x.numpy(), y.numpy())
plt.show()
```



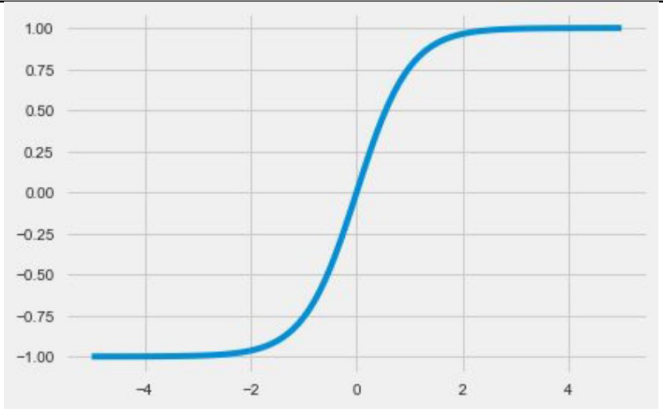
As you can observe from the plot, the sigmoid function saturates (i.e., produces extreme valued outputs) very q uickly and for a majority of the inputs. This can become a problem because it can lead to the gradients becoming either zero or diverging to an overflowing floating-point value. These phenomena are also known as *vanishing gradient problem* and *exploding gradient problem*, respectively. As a consequence, it is rare to see sigmoid units used in neural networks other than at the output, where the squashing property allows one to interpret outputs as probabilities.

b. **Tanh:** The tanh activation function is a cosmetically different variant of the sigmoid. This becomes clear when you write down the expression for tanh:

$f(x)=\tanh x= (e^{x} – e^{-x}) / (e^{x} + e^{-x})$
With a little bit of wrangling (which we leave for you as an exercise), you can convince yourself that tanh is simply a linea r transform of the sigmoid function. Notice that tanh, like the sigmoid, is also a "squashing" function, except that it maps the set of real values from $(–\infty, +\infty)$ to the range [-1, +1].

```
import torch
import matplotlib.pyplot as plt

x = torch.range(-5., 5., 0.1)
y = torch.tanh(x)
plt.plot(x.numpy(), y.numpy())
plt.show()
```

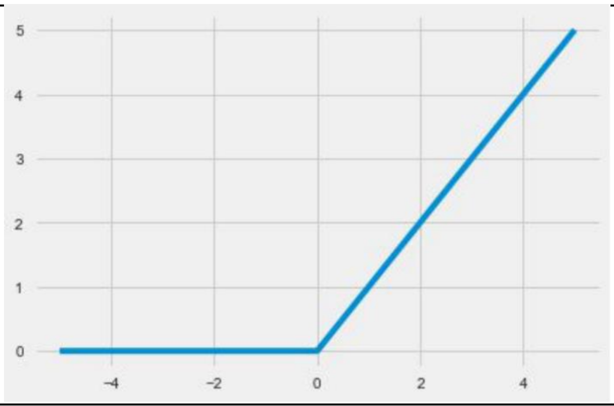


c. **ReLU:** stands for rectified linear unit.
f(x)=max(0,x)
So, all a ReLU unit is doing is clipping the negative values to zero.

```
import torch
import matplotlib.pyplot as plt

relu = torch.nn.ReLU()
x = torch.range(-5., 5., 0.1)
y = relu(x)

plt.plot(x.numpy(), y.numpy())
plt.show()
```

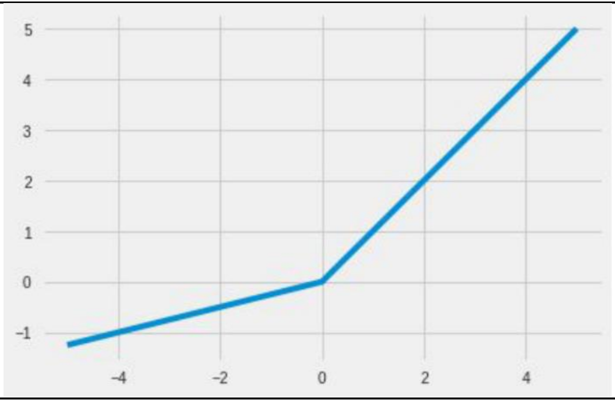


The clipping effect of ReLU that helps with the vanishing gradient problem can also become an issue, where over time certain outputs in the network can simply become zero and never revive again. This is called the "dying ReLU" problem. To mitigate that effect, variants such as the Leaky ReLU and Parametric ReLU (PReLU) activation functions have proposed, where the leak coefficient a is a learned parameter.
f(x)=max(x,ax)

```
import torch
import matplotlib.pyplot as plt

prelu =
torch.nn.PReLU(num_parameters=1)
x = torch.range(-5., 5., 0.1)
y = prelu(x)

plt.plot(x.numpy(), y.numpy())
plt.show()
```



d. **Softmax:** Like the sigmoid function, the softmax function squashes the output of each unit to be between 0 and 1. However, the softmax operation also divides each output by the sum of all the outputs, which gives us a discrete probability distribution3 over k possible classes:

softmax(xi) = $e^{x_i} / \Sigma^{k}_{j=1} e^{x_j}$
The probabilities in the resulting distribution all sum up to one. This is very useful for interpreting outputs for classification tasks, and so this transformation is usually paired with a probabilistic training objective, such as categorical cross entropy.

```
import torch.nn as nn
import torch

softmax = nn.Softmax(dim=1)
x_input = torch.randn(1, 3)
y_output = softmax(x_input)
print(x_input)
print(y_output)
print(torch.sum(y_output, dim=1))
```

Output:

```
tensor([[ 0.5836, -1.3749, - 1.1229]])
tensor([[ 0.7561,  0.1067,  0.1372]])
tensor([ 1.])
```

Loss Functions: Loss function takes a truth (y) and a prediction (ŷ) as an input and produces a real-valued score. The higher this score, the worse the model's prediction is. PyTorch implements more loss functions in its nn package.

a. Mean Squared Error Loss:
For regression problems for which the network's output (ŷ) and the target (y) are continuous values, one common loss function is the mean squared error (MSE):

$$L_{MSE}(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y-\hat{y})^2$$

The MSE is simply the average of the squares of the difference between the predicted and target values. There are several other loss functions that you can use for regression problems, such as mean absolute error (MAE) and root mean squared error (RMSE), but they all involve computing a real-valued distance between the output and target.

```
import torch
import torch.nn as nn

mse_loss = nn.MSELoss()
outputs = torch.randn(3, 5,
requires_grad=True)
targets = torch.randn(3, 5)
loss = mse_loss(outputs, targets)
print(loss)
```

Output:

```
tensor(3.8618)
```

b. **Categorical Cross-Entropy Loss:** The categorical cross-entropy loss is typically used in a multiclass classification setting in which the outputs are interpreted as predictions of class membership probabilities. The target (y) is a vector of n elements that represents the true multinomial distribution[4] over all the classes. If only one class is correct, this vector is a one-hot vector.

$$L_{cross\_entropy}\left(y,\hat{y}\right) = -\sum_i y_i \log\left(\hat{y}_i\right)$$

Cross-entropy and the expression for it have origins in information theory, but for the purpose of this section it is helpful to consider this as a method to compute how different two distributions are. We want the probability of the correct class to be close to 1, whereas the other classes have a probability close to 0.

To correctly use PyTorch's CrossEntropyLoss() function, it is important to understand the relationship between the network's outputs, how the loss function is computed, and the kinds of computational constraints that stem from really representing floating-point numbers. Specifically, there are four pieces of information that determine the nuanced relationship between network output and loss function. First, there is a limit to how small or how large a number can be. Second, if input to the exponential function used in the softmax formula is a negative number, the resultant is an exponentially small number, and if it's a positive number, the resultant is an exponentially large number. Next, the network's output is assumed to be the vector just prior to applying the softmax function.[5] Finally, the log function is the inverse of the exponential function,[6] and log(exp(x)) is just equal to x. Stemming from these four pieces of information, mathematical simplifications are made assuming the exponential function that is the core of the softmax function and the log function that is used in the cross-entropy computations in order to be more numerically stable and avoid really small or really large numbers. The consequences of these simplifications are that the network output without the use of a softmax function can be used in conjunction with PyTorch's CrossEntropyLoss() to optimize the probability distribution. Then, when the network has been trained, the softmax function can be used to create a probability distribution.

```
import torch
import torch.nn as nn

ce_loss = nn.CrossEntropyLoss()
outputs = torch.randn(3, 5, requires_grad=True)
targets = torch.tensor([1, 0, 3],
dtype=torch.int64)
loss = ce_loss(outputs, targets)
print(loss)
```

Output:

```
tensor(2.7256)
```

In this code example, a vector of random values is first used to simulate network output. Then, the ground truth vector, called targets, is created as a vector of integers because PyTorch's implementation of CrossEntropyLoss() assumes that each input has one particular class, and each class has a unique index. This is why targets has three elements: an index representing the correct class for each input. From this assumption, it performs the computationally more efficient operation of indexing into the model output.

c. **Binary Cross-Entropy Loss:** For binary classification.
We create a binary probability output vector, probabilities, using the sigmoid activation function on a random vector that represents the output of the network. Next, the ground truth is instantiated as a vector of 0's and 1's. Finally, we compute binary cross-entropy loss using the binary probability vector and the ground truth vector.

```
bce_loss = nn.BCELoss()
sigmoid = nn.Sigmoid()
probabilities = sigmoid(torch.randn(4, 1,
requires_grad=True))
targets = torch.tensor([1, 0, 1, 0],
dtype=torch.float32).view(4, 1)
loss = bce_loss(probabilities, targets)
print(probabilities)
print(loss)
```

Output:

```
tensor([[ 0.1625],
        [ 0.5546],
        [ 0.6596],
        [ 0.4284]])
tensor(0.9003)
```