T05460301-01

Deep Learning (Fall 2023)

Ikbeom Jang

ijang@hufs.ac.kr

CES HUFS

Multilayer Perceptrons

- **Multilayer Perceptrons**

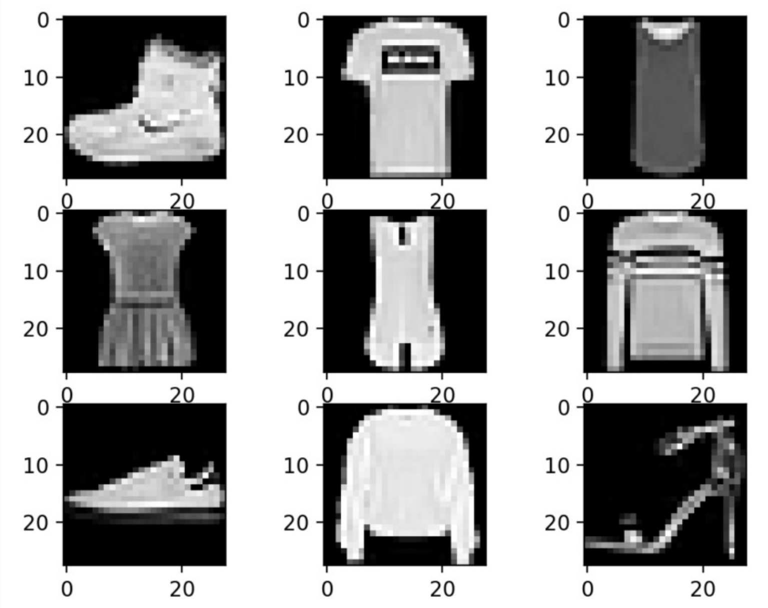
Multilayer Perceptrons

- We introduced softmax regression for recognizing 10 clothing categories from low-resolution images

- The previous section covered implementing the algorithm from scratch and using high-level APIs

- It taught data wrangling, probability distribution, loss function application, and parameter optimization.

- This knowledge paves the way for exploring deep neural networks, the primary focus of the course.



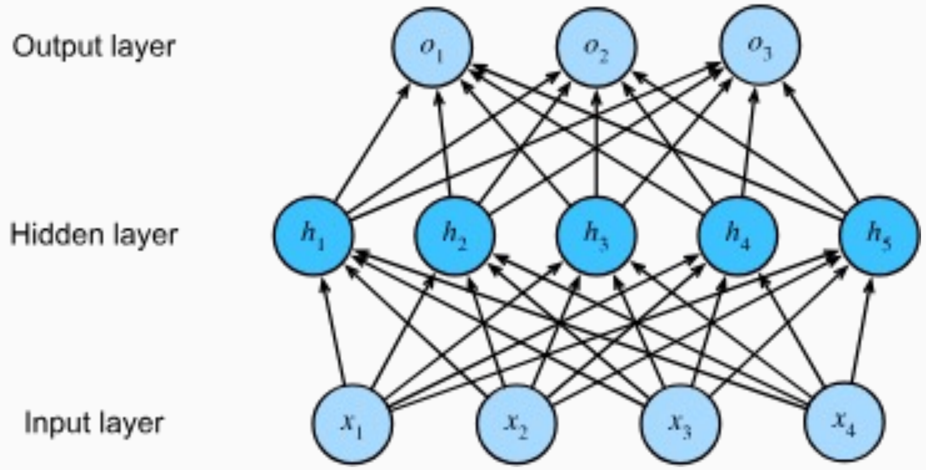
```
import torch
from d2l import torch as d2l
```

Hidden Layers

- Affine transformations were introduced as linear transformations with added bias.
- The model architecture for softmax regression involves mapping inputs to outputs using a single affine transformation followed by a softmax operation.
- This approach works when labels are related to input data through a simple affine transformation.
- However, assuming linearity, particularly in affine transformations, is a strong assumption.
- This indicates that more complex, non-linear models may be needed when the relationship between inputs and labels is not simple.

Hidden Layers

- To overcome the limitations of linear models, we can add **hidden layers**, commonly achieved by **stacking fully connected layers**.
- Each layer feeds its output into the layer above it until we generate the final outputs.
- In this architecture, the first (L-1) layers act as representations, and the final layer serves as the linear predictor.
- This design is known as a **multilayer perceptron (MLP)**, typically with an input layer, hidden layers, and an output layer.
- The described MLP has four inputs, five hidden units in its hidden layer, and three outputs, resulting in two layers.
- All layers in the MLP are fully connected, allowing each input to influence every neuron in the hidden layer, and each of those influences every neuron in the output layer.



Hidden Layers

For a single-hidden-layer MLP, we can calculate its output $\boldsymbol{O}$ as follows:

$$\boldsymbol{H} = \boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}$$
$$\boldsymbol{O} = \boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}$$

where

$$\boldsymbol{X} \in R^{n\times d}$$
$$\boldsymbol{H} \in R^{n\times h}$$
$$\boldsymbol{W}^{(\mathbf{1})} \in R^{d\times h}$$
$$\boldsymbol{b}^{(\mathbf{1})} \in R^{1\times h}$$
$$\boldsymbol{W}^{(\mathbf{2})} \in R^{h\times q}$$
$$\boldsymbol{b}^{(\mathbf{2})} \in R^{1\times q}$$
$$\boldsymbol{O} \in R^{n\times q}$$

- After adding the hidden layer, our model now requires us to track and update additional sets of parameters.
- It is still a linear model!
    - An affine function of an affine function is itself an affine function.

Hidden Layers

- To realize the potential of multilayer architectures, we need one more key ingredient.
- It's a **nonlinear activation function $\boldsymbol{\sigma}$** to be applied to each hidden unit following the affine transformation.
- For instance, a popular choice is the ReLU (rectified linear unit) activation function is $\sigma(x) = \max(0, x)$
  - It operates on its arguments elementwise.
- The outputs of activation functions are called activations.
- In general, with activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\boldsymbol{H} = \sigma\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)$$
$$\boldsymbol{O} = \boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}$$

- Since each row in $\boldsymbol{X}$ corresponds to an example in the minibatch, we define the nonlinearity σ to apply to its inputs in a rowwise fashion, i.e., one example at a time.
  - We used the same notation for softmax.
  - Quite frequently the activation functions we use apply not merely rowwise but elementwise.
  - That means that after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units.

Hidden Layers

- Since each row in $\boldsymbol{X}$ corresponds to an example in the minibatch, we define the nonlinearity σ to apply to its inputs in a rowwise fashion, i.e., one example at a time.
  - We used the same notation for softmax.
  - Quite frequently the activation functions we use apply not merely rowwise but elementwise.
  - That means that after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units.

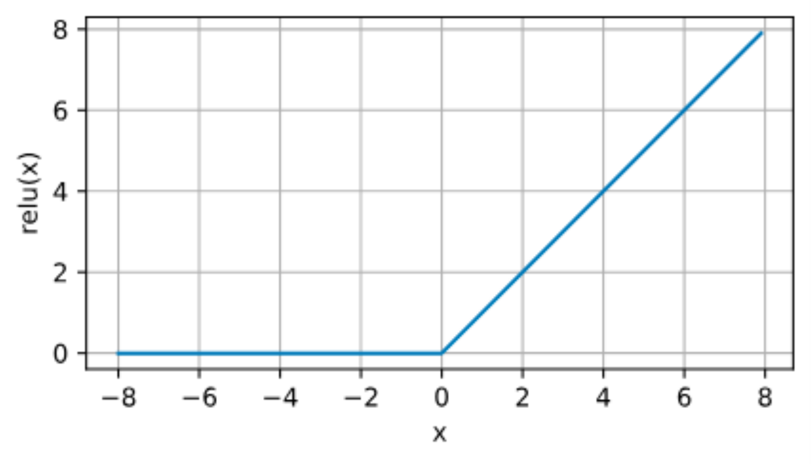
- To build more general MLPs, we can continue stacking such hidden layers one atop another, yielding ever more expressive models.
  - For example,
  - $\boldsymbol{H}^{(1)} = \sigma_1\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)$ and $\boldsymbol{H}^{(2)} = \sigma_2\left(\boldsymbol{H}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right)$

Activation Functions

- Activation functions decide whether a neuron should be activated or not, by calculating the weighted sum and further adding bias to it.
- They are differentiable operators for transforming input signals to outputs, while most of them add nonlinearity.
- Because activation functions are fundamental to deep learning, let's briefly survey some common ones.

- Rectified linear unit (ReLU)
- $ReLU(x) = \max(0, x)$

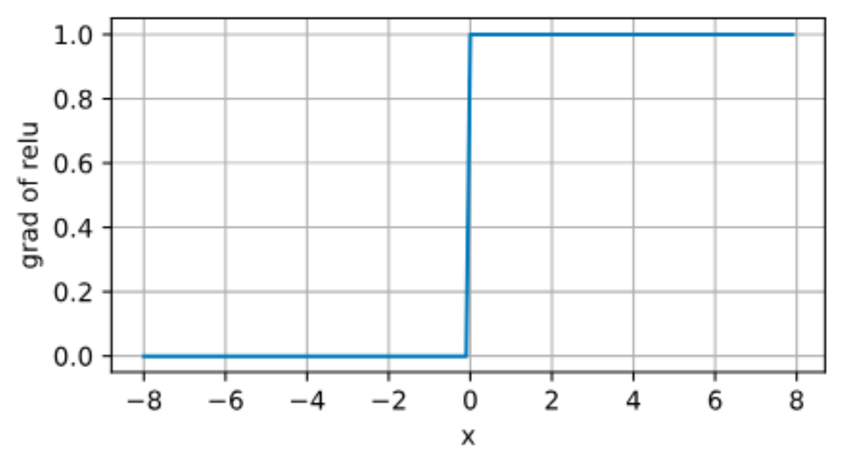
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



Activation Functions

- The ReLU function retains only positive elements and discards all negative elements by setting the corresponding activations to 0.
- As you can see, the activation function is piecewise linear.
- Note that the ReLU function is not differentiable when the input takes value precisely equal to 0.
- In these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0.
- We can get away with this because the input may never actually be zero.

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



Activation Functions

- The reason for using ReLU is that its derivatives are particularly well behaved:
  - either they vanish or they just let the argument through.

- This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradients that plagued previous versions of neural networks

Activation Functions

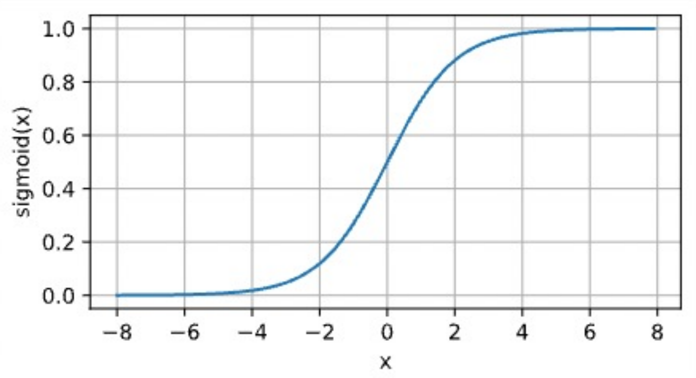
- Try also Sigmoid and Tanh functions.

$$sigmoid(x) = \frac{1}{1 + \exp(-x)}$$

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

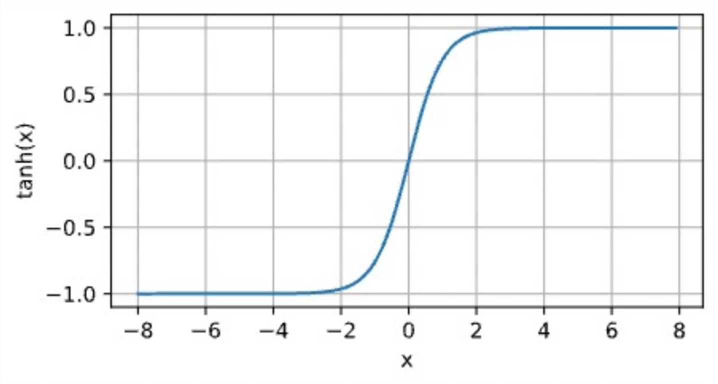
PYTORCH MXNET JAX TENSORFLOW

```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



PYTORCH MXNET JAX TENSORFLOW

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



Lab

- Implement MLP while referencing the codes provided