Lecture 7

Feedforward Neural Networks

Contents

Artificial neural networks (ANNs) are simplified mathematical analogs of biological neural networks and form the foundation of modern machine learning. ANNs are powerful due to two main properties:

- **High expressiveness:** Neural networks can approximate a wide range of functions. With sufficient depth and width, they can capture complex patterns in data that simpler models cannot.
- **Flexibility and trainability:** Despite their complexity, neural networks can be trained efficiently using modern architectures and tools such as automatic differentiation.

Common types of neural networks include:

- *Feedforward Neural Networks (FNNs)* – the basic type, suitable for prediction tasks.
- *Recurrent Neural Networks (RNNs)* – designed for sequential or time-series data.
- *Graph Neural Networks (GNNs)* – operate on graph-structured data, aggregating information from neighboring nodes.
- *Transformer Networks* – handle long-range dependencies in sequences using self-attention mechanisms.
- *Generative Adversarial Networks (GANs)* – generate realistic data through adversarial training of a generator and discriminator.

In this lecture, we focus on the **feedforward neural network (FNN)**. FNNs are fully connected networks in which information flows in only one direction—from the input layer, through any hidden layers, to the output layer—without cycles or loops. Understanding FNNs provides a foundation for

more complex neural network architectures.

The linear regression and logistic regression models we studied earlier can be seen as simple examples of FNNs.

1 Recap: Linear Regression and Logistic Regression

In Lecture 5, we introduced two basic linear models in machine learning: linear regression and logistic regression.

Linear Regression

Given feature-label pairs $(\mathbf{x}_k, y_k)$ in a training dataset, linear regression assumes that the label y_k can be approximated as a *linear* combination of features:

$$y_k \approx f_{\boldsymbol{\theta}}(\mathbf{x}_k) = \theta_0 + \theta_1 x_{k1} + \theta_2 x_{k2} + \cdots + \theta_p x_{kp} = \theta_0 + \sum_{i=1}^{p} \theta_i x_{ki}, \tag{1}$$

where p is the number of features and $\boldsymbol{\theta} = (\theta_0, \theta_1, \ldots, \theta_p)$ are the **parameters** of the model. Here, x_{ki} is the i-th feature of the k-th data point.

To optimize $\boldsymbol{\theta}$, we minimize a **loss function** that measures the difference between the true label y_k and the predicted value $f_{\boldsymbol{\theta}}(\mathbf{x}_k)$:

$$\text{Loss}(\boldsymbol{\theta}) = \sum_{k=1}^{N} (y_k - f_{\boldsymbol{\theta}}(\mathbf{x}_k))^2. \tag{2}$$

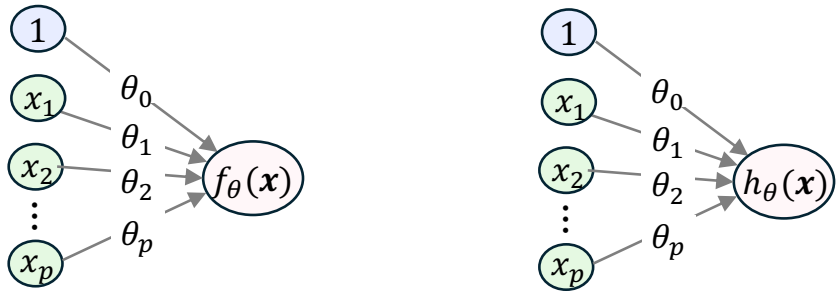
The above loss is called the mean squares error (MSE). The optimal parameters are those that minimize the loss:

$$\boldsymbol{\theta}_{\text{opt}} = \arg\min_{\boldsymbol{\theta}} \text{Loss}(\boldsymbol{\theta}). \tag{3}$$

This optimization strategy is called the **least squares method**, and $\text{Loss}(\boldsymbol{\theta})$ is referred to as the least squares error.

Logistic Regression

Logistic regression extends linear regression to *classification*, where the label y is discrete, e.g., $y = 0$ or $y = 1$.



(a) Linear Regression (b) Logistic Regression

Figure 2: Neural network representations of linear regression and logistic regression.

Starting from the linear model $f_{\boldsymbol{\theta}}(\mathbf{x}_k)$, we apply the **sigmoid function**:

$$h_{\boldsymbol{\theta}}(\mathbf{x}_k) = \frac{1}{1 + \exp\left(-f_{\boldsymbol{\theta}}(\mathbf{x}_k)\right)}. \tag{4}$$

Figure 1 illustrates the relationship between $h_{\boldsymbol{\theta}}(\mathbf{x}_k)$ and $f_{\boldsymbol{\theta}}(\mathbf{x}_k)$. The output $h_{\boldsymbol{\theta}}(\mathbf{x}_k)$ is bounded between 0 and 1, allowing it to be interpreted as a probability. Specifically, $h_{\boldsymbol{\theta}}(\mathbf{x}_k)$ gives the probability that $y = 1$, while $1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k)$ gives the probability that $y = 0$.

The sigmoid function introduces **non-linearity** into the model.

The loss function for logistic regression is the cross-entropy between y_k and $h_{\boldsymbol{\theta}}(\mathbf{x}_k)$:

$$\text{Loss}(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{k=1}^{n}\left[y_k \ln h_{\boldsymbol{\theta}}(\mathbf{x}_k) + (1 - y_k)\ln(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k))\right]. \tag{5}$$

This loss is preferred because it simplifies gradient computation compared to the least squares loss.

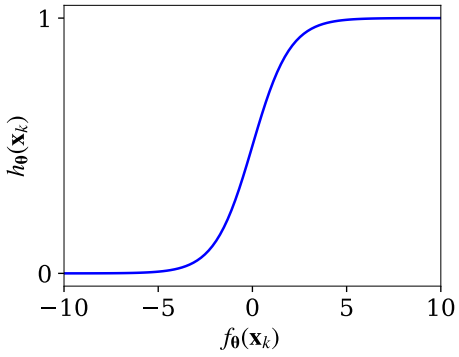


Figure 1: The sigmoid function.

1.1 Connections to Neural Networks

Linear and logistic regressions can be represented as simple neural networks, as shown in Fig. 2.

Figure 2 depicts the simplest neural networks with only two layers: the *input* layer and the *output* layer.

- **Input layer**: features $\mathbf{x}$.

- **Output layer**: predicted label values.
- **Edges (weights)**: each connection has a weight θ_k.
- **Bias**: θ_0 acts as the bias term.
- **Activation function**: applying the sigmoid function on $f_{\boldsymbol{\theta}}(\mathbf{x}_k)$ introduces non-linearity, acting as an activation function.

A neural network is nothing but a mathematical form that provides a more complicated model than the statistical models we discussed.

2 The Architecture

A typical **feed-forward** neural network has the structure shown in Fig. 3, composed of the following parts:

Layers:

- Input layer: the features of the dataset.
- Hidden layers: intermediate computational layers.
- Output layer: produces the prediction (scalar or vector).

Nodes and Edges:

- Nodes: The nodes in a neural network is called a **neuron**.
- Edges: edges carry weights and connect neurons between adjacent layers.

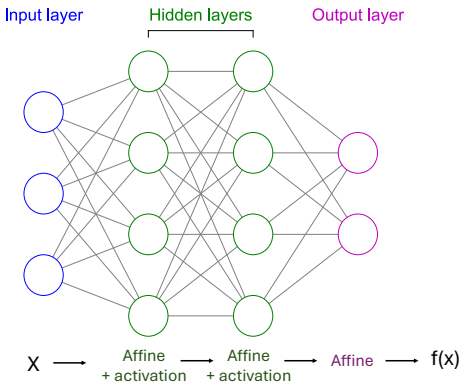


Figure 3: The structure of the feed-forward neural network.

Feed-Forward Mapping:

Here feed-forward means that the computation is done from left to right:

$$\text{input layer} \longrightarrow \text{hidden layer 1} \longrightarrow \text{hidden layer 2} \longrightarrow \cdots \longrightarrow \text{output layer}$$

2.1 Evaluating the Neurons

- Neurons store **activation values**.
- Values are **passed through layers** via weighted edges.

The value of a neuron is evaluated from the previous layer based on the following two operations:

1. Affine transformation.

$$\tilde{\mathbf{z}}^{(l)} = W^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}, \tag{6}$$

where $\mathbf{z}^{(l-1)}$ are the neuron values of the $(l-1)$th layer, $W^{(l)}$ is the **weight** matrix for the l-th layer, and $\mathbf{b}^{(l)}$ is the **bias** for the l-th layer. Weight and bias (**W&B**) are the parameters of a neural network.

The kth element in $\tilde{\mathbf{z}}^{(l)}$ is

$$\tilde{z}_k^{(l)} = \sum_{j=1}^{n_{l-1}} W_{kj}^{(l)} z_j^{(l-1)} + b_k^{(l)}, \tag{7}$$

We can see that each neuron in the lth layer is equivalent to the output in the linear regression neural network in Fig. 2.

2. Activation.

Activation adds *non-linearity* to the neural network.

$$\mathbf{z}^{(l)} = \phi(\tilde{\mathbf{z}}^{(l)}), \tag{8}$$

which means for each element,

$$z_k^{(l)} = \phi(\tilde{z}_k^{(l)}) \tag{9}$$

The sigmoid function is an example of an activation function.

Activation functions are essential in neural networks. Without activation, all layers of a network collapse into a single affine transformation, regardless of depth.

Common activation functions include (see Fig. 4): 1) sigmoid, 2) ReLu, 3) tanh, 4) softmax. Among these, **ReLu** is the industry standard for hidden layers, while sigmoid, tanh, softmax are for output layers (mainly for classification).

There are also variants of ReLu, such as Leaky ReLU, ELU, and GELU.

In summary, the feed-forward mapping for each layer is

$$\mathbf{z}^{(l)} = \phi\left(W^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right). \tag{10}$$

All we are doing is a **sequential** operation:

$$\mathrm{X} \rightarrow \text{affine} \rightarrow \text{activation} \rightarrow \text{affine} \rightarrow \text{activation} \cdots \rightarrow y$$

2.2 Expressiveness

Why is the neural network so powerful? One reason is that we can make it arbitrarily expressive by altering the following two settings

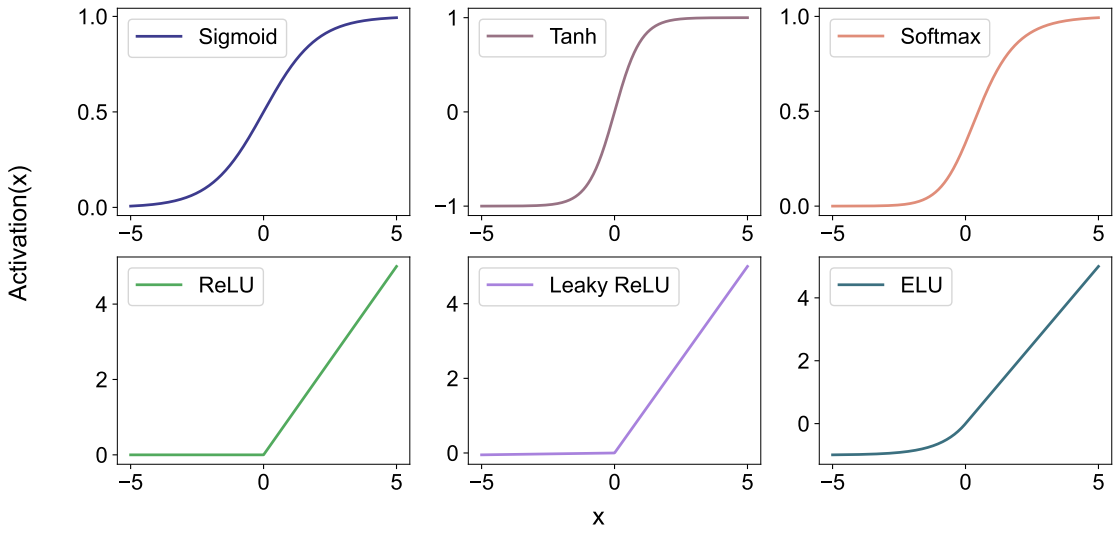


Figure 4: Common activation functions.

- The number of hidden layers and hidden neurons.
- Affine and activation transformations. For each neuron, we add a complicated non-linear transformation.

Combining the above two components, we can make a neural network arbitrarily expressive, i.e., can approximate any continuous functions. This is governed by the **universal approximation theorem**.

3 Training a FNN

Building and training a neural network involves the following steps

1. Define the architecture, including
   - Number of layers.
   - Number of neurons per layer.
   - Activation function.
2. Initialize W&B parameters.
3. Select a loss function and an optimizer.
4. Start the training loop.

3.1 Loss and Gradient

Although neural networks can be made arbitrarily expressive, their parameters must remain *optimizable* in practice. A key advantage of neural network architectures is that they are naturally

paired with efficient, *gradient-based* optimization frameworks.

The **parameters** of a neural network model consist of its *weights* and *biases* (W&B),

$$\boldsymbol{\theta} = \left\{W^{(l)}, \mathbf{b}^{(l)}\right\}, \tag{11}$$

where $W^{(l)}$ is the weight matrix of the l-th layer and $\mathbf{b}^{(l)}$ is the corresponding bias vector.

The goal of training is to optimize the W&B parameters by minimizing a **loss function** $\mathrm{Loss}(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}_{\mathrm{opt}} = \arg\min_{\boldsymbol{\theta}} \mathrm{Loss}(\boldsymbol{\theta}). \tag{12}$$

A common form of the loss function is the empirical average over the training dataset,

$$\mathrm{Loss}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{k=1}^{N} \mathrm{loss}(y_k, f_{\boldsymbol{\theta}}(\mathbf{x}_k)), \tag{13}$$

where $\mathrm{loss}(y_k, f_{\boldsymbol{\theta}}(\mathbf{x}_k))$ measures the discrepancy between the true label y_k and the model prediction $f_{\boldsymbol{\theta}}(\mathbf{x}_k)$.

The choice of loss function depends on the specific machine learning task. In general, a suitable loss function should satisfy two basic requirements:

- It penalizes discrepancies between predictions and targets.
- It is efficient to minimize using gradient-based optimization.

So far, we have encountered two commonly used loss functions: mean squared error (MSE) and cross entropy. As we study more machine learning models, we will encounter additional loss functions designed for different tasks.

Common loss functions are defined in standard ML libraries. For example, with `PyTorch`:

- MSE: `torch.nn.MSELoss()`.
- Cross-entropy: `torch.nn.CrossEntropyLoss()`.

The loss function is minimized by gradient-based methods, more precisely, the **gradient descent** methods, where the parameters are updated using the gradient of the loss

$$\boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i - \eta \nabla_{\boldsymbol{\theta}} \mathrm{Loss}(\boldsymbol{\theta}_i), \tag{14}$$

where i is the optimization step, and η is a constant that controls the step size, called the **learning rate**.

3.2 Forward Pass and Backpropagation

Training a neural network requires evaluating the loss and gradient, which depend on the *forward pass* and *backpropagation*, respectively.

- Forward pass computes the model output and then the loss value.
- Backpropagation computes the gradients of the loss with respect to the parameters.

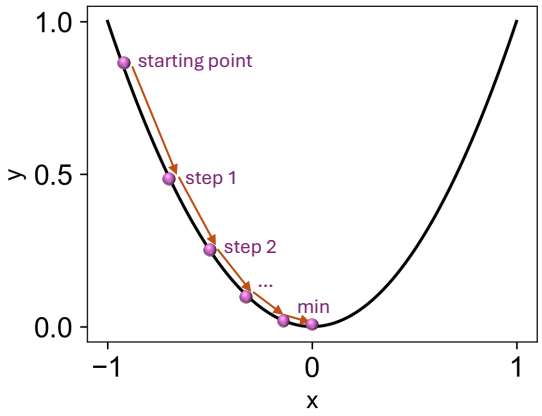


Figure 5: Illustration of gradient descent.

1. Forward Pass

For the l-th layer,

$$z_k^{(l)} = \phi(\tilde{z}_k^{(l)}), \tag{15}$$

and after we pass all the layers, we get the output $f_{\boldsymbol{\theta}}(\mathbf{x})$, and the loss function

$$\text{Loss}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{k=1}^{n} \text{loss}(y_k, f_{\boldsymbol{\theta}}(\mathbf{x}_k)). \tag{16}$$

2. Backpropagation

Backpropagation starts at the output layer, computes the gradient there, and then propagates the error backward through each layer, accumulating gradients for all weights and biases.

$$\frac{\partial \text{Loss}}{\partial W^{(l)}}, \quad \frac{\partial \text{Loss}}{\partial \mathbf{b}^{(l)}}. \tag{17}$$

The implicit implementation of backpropagation is not required. Usually, you can choose an **optimizer** in ML libraries.

Example: 3-Layer FNN

Consider a neural network with one hidden layer and activation function $\phi(\cdot)$. The model parameters are $\{W^{(1)}, \mathbf{b}^{(1)}, W^{(2)}, \mathbf{b}^{(2)}\}$, shown in Fig. 6.

1. Forward Pass. The hidden-layer activation is first computed as

$$\mathbf{z}^{(1)} = \phi\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right).$$

The output of the network is then evaluated by

$$f_\theta(\mathbf{x}) = W^{(2)}\mathbf{z}^{(1)} + \mathbf{b}^{(2)}.$$

Substituting $\mathbf{z}^{(1)}$, we obtain

$$f_\theta(\mathbf{x}) = W^{(2)}\left[\phi\left(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)\right] + \mathbf{b}^{(2)}.$$

Therefore, the evaluation order is

$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} \longrightarrow f_\theta(\mathbf{x}),$$

which proceeds from **left to right**. This is called the **forward pass**.

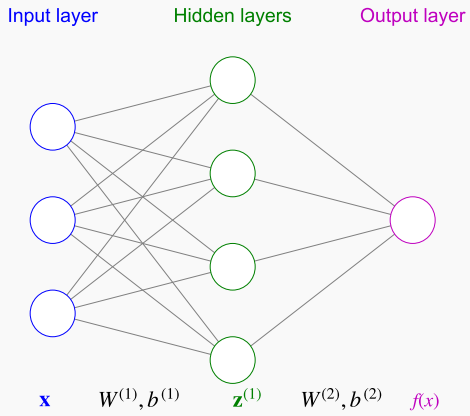


Figure 6: A 3-Layer FNN.

2. Backpropagation. To train the network, we compute gradients of the loss function using the chain rule. For the first-layer weights,

$$\frac{\partial \text{Loss}}{\partial W^{(1)}} = \frac{\partial \text{Loss}}{\partial f_\theta(\mathbf{x})} \frac{\partial f_\theta(\mathbf{x})}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial W^{(1)}}.$$

Similarly, for the first-layer bias,

$$\frac{\partial \text{Loss}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \text{Loss}}{\partial f_\theta(\mathbf{x})} \frac{\partial f_\theta(\mathbf{x})}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{b}^{(1)}}.$$

Both gradients share the common factors

$$\frac{\partial \text{Loss}}{\partial f_\theta(\mathbf{x})} \quad \text{and} \quad \frac{\partial f_\theta(\mathbf{x})}{\partial \mathbf{z}^{(1)}}$$

which originate from the *later layers* of the network.

Backpropagation exploits this structure by first computing gradients for the output layer and then propagating them backward to earlier layers. Thus, gradient evaluation proceeds from **right to left**, which motivates the name **backpropagation**.

Extra Reading: A closely related idea to backpropagation is *automatic differentiation*. It is a powerful and elegant numerical framework for computing derivatives efficiently. I have uploaded a separate file on Canvas that explains automatic differentiation in more detail.

3.3 Choosing an Optimizer

In practice, you do not need to implement gradient descent from scratch — you can simply choose an **optimizer**.

A widely used optimizer in modern machine learning is **Adam** [1], which is a variant of stochastic gradient descent (SGD). The detailed workings of Adam are beyond the scope of this lecture. For now, it is sufficient to know that it is a gradient-based optimization method.

For all gradient descent methods, the most important hyperparameter is the **learning rate**.

PyTorch Example:

```
from torch import optim

# Create an Adam optimizer to update all learnable parameters
optimizer = optim.Adam(model.parameters(), lr=1e-3)
# model.parameters() specifies which tensors to optimize
# lr is the learning rate
```

`optim.SGD()` is another standard SGD optimizer.

3.4 Hyperparameters

In addition to the weight and bias parameters, building and training a neural network relies on setting many hyperparameters. Hyperparameters are settings chosen *before training* that control the architecture, learning process, and generalization of a neural network.

In the previous lectures and this lecture, we learned several hyperparameters:

- Structure related: Number of layers (depth), number of neurons per layer (width of each layer), and the activation function.
- Optimization related: regularization factor λ and learning rate η.

We will learn two more important hyperparameters: **batch size** and **number of epochs**.

Batch Size.

When the dataset is large, it is memory-inefficient to pass all data into the optimization once. Instead, we divide the whole dataset into several batches, passed to the optimization in sequence. The `batch size` is the size of each batch.

Number of epochs.

Once we pass all batches to the optimization, the whole dataset is trained, we call it an **epoch.** We train several epochs to make sure the model is optimized.

Therefore, the training loop has two layers, shown in Algorithm 1.

Algorithm 1 Neural Network Training Loop

```
Training dataset 𝒟, number of epochs L_E, batch size N_B
for epoch = 1 to L_E do                                  ▷ Outer loop over epochs
    for each mini-batch (X_b, y_b) of size N_B in 𝒟 do    ▷ Inner loop over batches
        Forward pass: compute predictions
        Compute loss: Loss(θ)
        Backward pass: compute gradients
        Update network parameters: θ ← θ − η∇_θ Loss(θ)
    end for
end for
```

3.4.1 Tuning Hyperparameters

Successfully training a neural network requires careful tuning of its hyperparameters. Here we provide guidelines for tuning the key hyperparameters discussed earlier.

Depth and width. Depth (number of layers) and width (number of neurons per layer) determine the complexity of the network.

- **Model capacity:** Increasing depth allows the network to learn hierarchical features, while increasing width enables it to detect more parallel patterns.
- **Overfitting vs. underfitting:** Excessive capacity can overfit small datasets, whereas too little capacity prevents the model from capturing the underlying structure.
- **Vanishing/exploding gradients:** Extremely deep networks are harder to train without specialized architectures or initialization techniques.

Activation function. Activation functions allow the network to learn non-linear relationships.

- **ReLU** is the standard choice for hidden layers. It avoids the vanishing gradient problem and introduces sparsity, since negative values are set to zero and not passed to the next layer.
- **Sigmoid** or **Softmax** are typically used only in output layers for classification tasks.

Learning rate. The learning rate is arguably the most important hyperparameter.

- **Convergence speed:** A rate that is too high can cause divergence, while a rate that is too low slows training.
- **Decay schedules:** Gradually lowering the learning rate during training can help the model settle into a sharp minimum.

Batch size.

- **Representation:** Very small batches may poorly represent the dataset.
- **Stochasticity:** Small batches introduce gradient noise, which can help escape local minima.
- **Memory and speed:** Larger batches exploit GPU parallelism but require more memory.
- **Generalization:** Very large batches can sometimes reduce generalization performance.

Number of epochs.

- **Convergence:** Training should continue until the loss flattens out.
- **Early stopping:** To avoid overfitting, stop training when validation loss stops improving, even if training loss continues to decrease.

4 PyTorch Realization

From this point, we will move beyond the simple built-in ML models in `scikit-learn` and build our own models based on neural networks.

We first introduce a very powerful data structure: Python *classes*, and then walk through the implementation of building and training a neural network with `PyTorch`.

4.1 The Simplest Example

We can use `torch.nn.sequential()` to serialize the layered operations (affine → activation → affine → activation $\cdots$).

```
import torch.nn as nn

# Minimal feedforward network for 4 layers.
model = nn.Sequential(
    nn.Linear(input_size, hidden_size_1), # affine
    nn.ReLU(), # activation
    nn.Linear(hidden_size_1, hidden_size_2) # affine
    nn.ReLU(), # activation
    nn.Linear(hidden_size_2, output_size) # affine
    # nn.Sigmoid() # activation of output layer is optional
)
```

We can make the NN more complicated by defining a *class*.

4.2 Python Classes

Before building neural networks in `PyTorch`, it is important to understand Python *classes*. Classes allow us to organize code by combining *data* and *functions* in a single object. This is

especially useful for neural networks, where we want to store weights, define forward passes, and implement training logic.

1. What is a class?

A class is a blueprint for creating **objects**, which are instances of the class. A class has two types of members:

- **Attributes:** data values stored in the object.
- **Methods:** functions that operate on the object's attributes.

Classes help us:

- Store data in a structured way (e.g., weights of a network)
- Define functions (methods) that operate on that data
- Reuse and extend code easily

2. Basic Syntax

```python
# Define a simple class
class SimpleClass:
    def __init__(self, arg1, arg2):  # constructor
        self.x = arg1
        self.y = arg2

    def func1(self):
        return self.x + self.y

    def func2(self, arg3):
        return self.func1() + arg3

# Using the class
my_obj = SimpleClass(0, 1)  # initialize an object
a = my_obj.func1()           # a = 0 + 1 = 1
b = my_obj.func2(2)          # b = a + 2 = 3
```

Key points:

- `__init__` is the constructor: it initializes the object's attributes (e.g., weights, biases).
- `self` refers to the object itself. Use it inside the class to access attributes or call methods.

- When using the object externally, you do not include `self`; only provide the method's arguments.
- Objects allow us to encapsulate both data (attributes) and behavior (methods) in a single, organized structure.

Example: a single neuron

```
class Neuron:
    def __init__(self, weight, bias):
        self.weight = weight
        self.bias = bias

    def forward(self, x):
        return x * self.weight + self.bias

# Create an instance of Neuron
my_neuron = Neuron(weight=2.0, bias=0.5)
output = my_neuron.forward(3.0)  # output = 3*2 + 0.5 = 6.5
```

3. Inheriting a Class

A class can inherit attributes and methods from a parent class. The child class can overwrite or extend the parent's functionality.

```
class ChildClass(ParentClass):
    # define new or overridden attributes/methods here
```

Example:

```
class Neuron2(Neuron):
    def eval(self, x, y):
        return (self.forward(x) - y)**2

# Using Neuron2
my_neuron2 = Neuron2(weight=2.0, bias=0.5)  # inherited __init__ from Neuron
loss = my_neuron2.eval(3.0, 6.2)
```

4.3 A PyTorch Network

Now we can build a simple 3-layer FNN for prediction. Our custom network will inherit from `torch.nn.Module`.

```python
import torch
import torch.nn as nn

class SimpleNN(nn.Module):
    """
    A simple 3-layer feedforward neural network:
    input layer -> hidden layer -> output layer
    """
    def __init__(self, input_size, hidden_size):
        super().__init__()  # initialize nn.Module
        self.fc1 = nn.Linear(input_size, hidden_size)  # input -> hidden
        self.relu = nn.ReLU()                          # activation
        self.fc2 = nn.Linear(hidden_size, 1)          # hidden -> output

    def forward(self, x):
        # apply each component sequentially
        h = self.fc1(x)
        h = self.relu(h)
        y = self.fc2(h)
        return y
```

Now you have learned how to define basic ML models and build a neural network. You can use this foundation to tackle a wide range of prediction tasks.

4.4 GPU Support (Optional Reading)

4.4.1 Choose the GPU backend

Google Colab provides access to a small-scale GPU for free. To use the GPU, you need to explicitly change the runtime backend: Click the `Runtime` tab, then select `Change runtime type`. In the dialog, you can choose `CPU` (default), `GPU`, or `TPU` as the backend. After saving, the notebook will restart, and your selected backend will be active.

On computational clusters, you need to specify the GPU node usage in your job submission settings.

4.4.2 Check GPU availability

To verify GPU is available, run:

```
print("CUDA available:", torch.cuda.is_available())
```

If `True` is returned, your session has access to an NVIDIA GPU with CUDA (Compute Unified Device Architecture).

You can also inspect the GPU information by running the following in your notebook:

```
!nvidia-smi
```

macOS Users

Apple GPUs do not use CUDA; they use Metal. To check for GPU support on macOS, run:

```
print("MPS available:", torch.backends.mps.is_available())
print("MPS built:", torch.backends.mps.is_built())
```

4.4.3 Specifying GPU Usage

In `PyTorch`, if you do not specify a device, computations default to the CPU. To ensure your program uses a GPU, follow two steps:

1. **Specify the device at the beginning of your program:**

   ```
   device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
   ```

   If Apple GPUs may also be used, you can extend it:

   ```
   device = (
       torch.device("cuda") if torch.cuda.is_available()
       else torch.device("mps") if torch.backends.mps.is_available()
       else torch.device("cpu")
   )
   ```

2. **Move your objects to the selected device.**
   For any `PyTorch` object, e.g., `my_obj`, use:

   ```
   my_obj.to(device)  # move my_obj to the chosen device
   ```

   For tensors, you can also specify the device when creating them:

   ```
   x = torch.randn(5, 10, device=device)
   ```

   Important: You cannot perform operations on objects located on different devices. For example:

```
# Illegal example
model = torch.nn.Linear(10, 1)      # on CPU
x = torch.randn(5, 10).to("cuda")   # on GPU

output = model(x)  # will raise RuntimeError
```

The resulting error is:

```
RuntimeError: Tensor for argument weight is on cpu but expected on mps
```

5 Labs

We will practice using `PyTorch` to build and train FNNs.

- Lab 7-1: `https://colab.research.google.com/drive/1jtv-utSukFjeEYhaQrP9Mk0r3Uu6FLZl?usp=sharing`
- Lab 7-2 (GPU version): `https://colab.research.google.com/drive/1t9d58V7SVrHBKsEXCv9z17?usp=sharing`

Problems

1. Evaluate the gradient

$$\frac{\partial \text{Loss}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

for the least-squares loss function and the cross-entropy loss function, respectively.

2. What are the sizes of $W^{(l)}$ and $\mathbf{b}^{(l)}$, respectively?

3. In the gradient descent formula, identify the parameters and hyperparameters

$$\boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i - \eta \nabla_{\boldsymbol{\theta}} \text{Loss}(\boldsymbol{\theta}_i).$$

4. Prove that without the activation step, two layers of affine transformation is effectively a one-layer affine transformation.

References

[1] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.