Lecture 2 Notes

Deep Learning AI335

1 Deep Feedforward Networks

A deep feedforward network is a multi-layer perceptron. We have multiple layers of *neurons* each performing the same basic operation, with the goal of approximating a complex function. Unlike single layer perceptron, a feedforward network is capable of capturing non-linear data.

1.1 Task

We define some task T, which a model will learn to perform. Examples of ML tasks are: Regression, Classification, Anomaly Detection, Clustering, etc. This task is usually defined by some input-output relationship which can be represented as a function:

$$f : X \to Y, \tag{1}$$

where X is the input space and Y is the output space.

A sample is collected as $(\mathbf{x_i}, \mathbf{y_i})$ pairs that make up our training data $\mathcal{D}$.

1.2 Model

Our model will try to learn the (unknown) input-output relationship from the training data, by adjusting some parameters $\boldsymbol{\theta}$ based on a learning algorithm. The network can be then be represented as:

$$\hat{\mathbf{y}} = f(\mathbf{x}; \boldsymbol{\theta}), \tag{2}$$

where $\boldsymbol{\theta}$ are our learnable parameters.

Each neuron/perceptron performs the operation:

$$s = \mathbf{w}^\top \cdot \mathbf{x} + b, \tag{3}$$

$$h = \phi(s), \tag{4}$$

where

$\mathbf{x}$: Input vector ($\mathbf{x} \in \mathbb{R}^d$,
$\mathbf{w}$: Weight vector ($\mathbf{w} \in \mathbb{R}^d$),
b : Bias term,
ϕ : Activation function.

We build a layer of n neurons that transform the input vector $\mathbf{x}$ into another vector $\mathbf{h}$:

$$\mathbf{s} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}, \tag{5}$$

$$\mathbf{h} = \phi(\mathbf{s}), \tag{6}$$

where

$$\begin{aligned}
\mathbf{W}: &\ \text{Weight matrix } (\mathbf{W} \in \mathbb{R}^{n \times d}),\\
\mathbf{b}: &\ \text{Bias vector } (\mathbf{b} \in \mathbb{R}^{d}),\\
\mathbf{h}: &\ \text{Output vector } (\mathbf{h} \in \mathbb{R}^{n}),\\
\phi: &\ \text{Activation function (applied element-wise)},
\end{aligned}$$

A network is made up of several of these layers, each layer *feeding* it's output into the next layers input. The computation at each layer l (for $l = 1, 2, \ldots, L$) is:

$$\mathbf{s}^{(l)} = \mathbf{W}^{(l)} \cdot \mathbf{h}^{(l-1)} + \mathbf{b}^(l), \tag{7}$$

$$\mathbf{h}^{(l)} = \phi^{(l)}(\mathbf{s}^{(l)}), \tag{8}$$

where

$$\begin{aligned}
\mathbf{h}_0: &\ \text{Input vector } (\mathbf{x}),\\
\mathbf{W}^{(l)}: &\ \text{Weight matrix for layer } l \ (\mathbf{W}^{(l)} \in \mathbb{R}^{n^{(l)} \times n^{(l-1)}}),\\
\mathbf{b}^{(l)}: &\ \text{Bias vector for layer } l \ (\mathbf{b}^{(l)} \in \mathbb{R}^{n^{(l)}}),\\
\mathbf{h}^{(l)}: &\ \text{Output vector for layer } l \ (\mathbf{h}^{(l)} \in \mathbb{R}^{n^{(l)}}),\\
\phi^{(l)}: &\ \text{Activation function for layer } l.
\end{aligned}$$

So our function for the output $\mathbf{y}$ is essentially just a composite function, of all the operations being performed at each layer:

$$\hat{\mathbf{y}} = f(\mathbf{x}; \boldsymbol{\theta}) = h^{(l)}(h^{(L-1)}(\cdots(h_1(\mathbf{x}, \boldsymbol{\theta}_1), \cdots), \boldsymbol{\theta}^{(L-1)}), \boldsymbol{\theta}^{(l)}), \tag{9}$$

where $\boldsymbol{\theta}_i$ represents the learnable parameters at a layer i.

2 Training Neural Networks using Backpropagation

2.1 Loss Function

Before going into training, we need a function that quantifies how well the model is performing with the current parameters. This is called the loss or cost function.

It's useful to think of our final prediction as less of a function $\hat{\mathbf{y}} = f(\mathbf{x}; \boldsymbol{\theta})$ and more as a probability distribution $\Pr(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ that generates output $\mathbf{y}$ given an input vector $\mathbf{x}$ and parameters $\boldsymbol{\theta}$. Some values of $\boldsymbol{\theta}$ will give us a higher probability of generating our desired output $\mathbf{y}$ while others will give us a lower probability.

So our training task now becomes: find the parameters $\boldsymbol{\theta}$ that maximize the likelihood of obtaining our desired outputs $\mathbf{y}_i$ given the inputs $\mathbf{x}_i$ in our training dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$.

Mathematically, this involves maximizing the joint likelihood over all training examples:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{N} \Pr(\mathbf{y}_i \mid \mathbf{x}_i; \boldsymbol{\theta}). \tag{10}$$

If we take the logarithm of the above equation we get the *log-likelihood*, which contains a sum instead of a product, which is more convenient:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \Pr(\mathbf{y}_i \mid \mathbf{x}_i; \boldsymbol{\theta}). \tag{11}$$

Instead of maximizing this likelihood, we typically frame this problem as minimizing a loss/cost function:

$$L(\boldsymbol{\theta}) = -\sum_{i=1}^{N} \log \Pr(\mathbf{y}_i \mid \mathbf{x}_i; \boldsymbol{\theta}). \tag{12}$$

This equation is known as *cross entropy loss*.

The sum here is essentially just finding the expected value of the log-likelihood over the training data. We can even rewrite the loss function as:

$$L(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{y},\mathbf{x} \sim \mathcal{D}} \log \Pr(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}). \tag{13}$$

So we can then define our optimization problem as:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, L(\theta). \tag{14}$$

Calculus and Linear Algebra Recap

Partial Diffrentiation

For a function $f(x_1, x_2, \cdots, x_n)$, the partial derivative with respect to x_i is denoted as:

$$\frac{\partial f}{\partial x_i} = \lim_{\Delta x_i \to 0} \frac{f(x_1, \ldots, x_i + \Delta x_i, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{\Delta x_i}. \tag{15}$$

$\frac{\partial f}{\partial x_i}$ represents the (instantaneous) rate of change of f with respect to x_i with all other variables held constant.

Chain Rule

For a composite function $z = f(u_1, u_2, \cdots, u_m)$, where each u_j if a function of $x_1, x_2, \cdots, x_n$:

$$u_j = g_j(x_1, x_2, \ldots, x_n), \quad j = 1, 2, \ldots, m. \tag{16}$$

The chain rule for the partial derivative of z with respect to x_i is:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial u_j} \frac{\partial u_j}{\partial x_i}. \tag{17}$$

The chain rule *aggregates* the effects x_i on z through all intermediate variables u_j.

Jacobian Matrix

For a vector valued function $f : \mathbb{R}^n \leftarrow \mathbb{R}^m$, defined as:

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ f_m(x_1, x_2, \ldots, x_n) \end{bmatrix}, \tag{18}$$

The Jacobian matrix $J_{\mathbf{f}}$ is an $m \times n$ matrix of first order partial derivatives:

$$J_{\mathbf{f}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{19}$$

Each element (i, j) of the Jacobian matrix represents the partial derivative of the i-th output with respect to the j-th input. The Jacobian captures how small changes in the input vector x affect the output vector $f(x)$.

Conditions for First-Order Differentiability

A function $f : \mathbb{R}^n \leftarrow \mathbb{R}$ is said to be *differentiable* at point x_0 if it can be well approximated by a linear function near that point. Formally, f is differentiable at x_0 if there exists a linear map L such that:

$$\lim_{\mathbf{h}\to\mathbf{0}} \frac{|f(\mathbf{x}_0 + \mathbf{h}) - f(\mathbf{x}_0) - L(\mathbf{h})|}{|\mathbf{h}|} = 0. \tag{20}$$

For first-order differentiability, the following conditions are often considered:

- *Existence of Partial Derivatives* - All partial derivatives $\frac{\partial f}{\partial x_i}$exist at x_0.
- *Continuity of Partial Derivatives* - The partial derivatives are continuous in a neighborhood around x_0.

Differentiability implies that the function has a well-defined tangent plane at x_0, and locally, the function behaves like its linear approximation.

2.2 Backpropagation Algorithm

To minimize the loss function $L(\theta)$ with respect to the parameters θ, we use gradient-based optimization algorithms like stochastic gradient descent (SGD) or its variants. These methods require the computation of the gradient of the loss function with respect to each parameter:

$$\nabla_\theta L(\theta). \tag{21}$$

To calculate this gradient we use the back propagation algorithm. The algorithm involves two passes through the network:

1. *Forward Pass* - Compute the outputs of the network for a given input $\mathbf{x}$ and store all intermediate activations $\mathbf{h}^{(l)}$ and pre-activations $\mathbf{s}^{(l)}$.
2. *Backward Pass* - Starting from the output layer, propagate the error backward through the network to compute the gradients with respect to each parameter.

For each layer l in the network, we compute:

$$\boldsymbol{\delta}^{(l)} = \left(W^{(l+1)^\top} \cdot \boldsymbol{\delta}^{(l+1)}\right) \odot \phi'(\mathbf{s}^{(l)}), \tag{22}$$

where $\odot$ denotes element-wise multiplication and $\phi'(\mathbf{s}^{(l)})$ is the derivative of the activation function evaluated at the pre-activation .

The gradient computation itself is:

$$\frac{\partial L}{\partial W^{(l)}} = \left(\mathbf{h}^{(l-1)}\right)^\top \cdot \boldsymbol{\delta}^{(l)}, \tag{23}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}, \tag{24}$$

Once the gradients are computed, we update the parameters using an optimization algorithm. For example, using stochastic gradient descent:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}, \tag{25}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \frac{\partial L}{\partial \mathbf{b}^{(l)}}, \tag{26}$$

where η is the learning rate.

Summary of Training Procedure

1. *Initialization* - Set initial values for weights and biases, typically using small random numbers.
2. *Forward Pass* - Compute the network output $\hat{\mathbf{y}}$ for input $\mathbf{x}$.
3. *Loss Computation* - Calculate the loss $L(\theta)$ using the appropriate loss function.
4. *Backward Pass* - Compute gradients $\nabla_\theta L(\theta)$ using backpropagation.
5. *Parameter Update* - Adjust the parameters θ using an optimization algorithm.
6. *Iteration* - Repeat steps 2–5 for multiple epochs over the training data.

Computational Graphs

Computational graphs are graphical representations of mathematical expressions where nodes correspond to operations or variables, and edges represent the flow of data between them.

Modern deep learning frameworks like TensorFlow and PyTorch utilize computational graphs to perform automatic differentiation and optimize computations on various hardware, such as GPUs and TPUs.

Computational Graphs in Neural Network Training

In the forward pass, inputs are fed into the computational graph, and computations proceed from the input nodes to the output nodes. Each operation in the graph processes its inputs and produces outputs, resulting in the final prediction of the model.

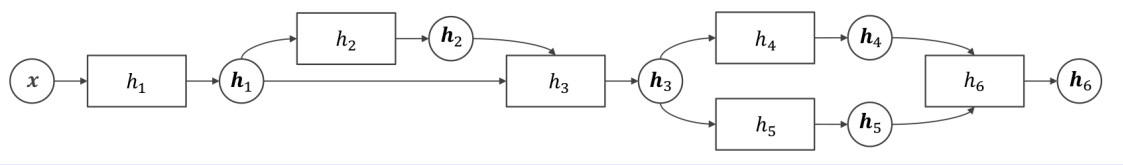

Figure 1: A simplified computational graph of a network.

During the backward pass, the computational graph is traversed in reverse to compute gradients of the loss function with respect to each parameter. This process utilizes the chain rule to efficiently compute derivatives (*automatic differentiation*).

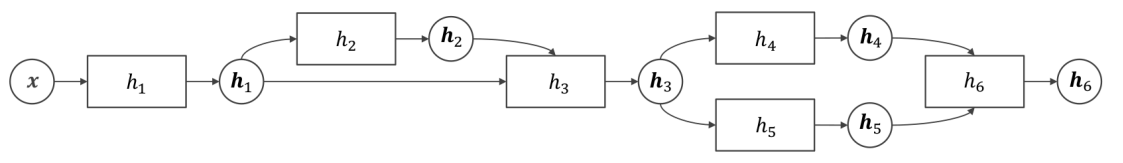

Figure 2: Reverse computational graph of the previous network.

3 Modular Architecture in Neural Networks

3.1 Module

A module is a self-contained component that performs a specific computation on its inputs to produce outputs. Modules can represent anything from a single neuron to an entire layer or even a collection of layers.

Note 1. We usually use the term module to refer to a set of layers with the same number of neurons and the same activation function.

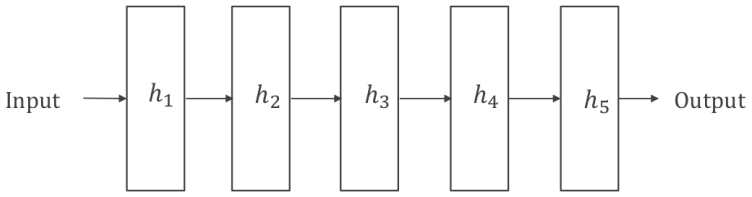

Figure 3: A simple feedforward network architecture represented using modules. Here each module is a layer. The activation for each layer is also a module since it *performs a specific computation on its inputs to produce outputs* but it is not shown in this diagram.

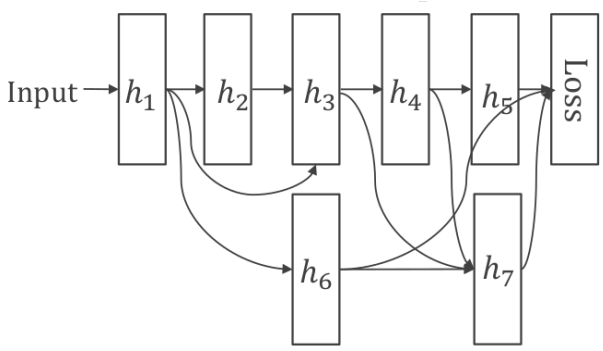

Figure 4: A more complex architecture utilizing *interweaved* and *skip* connections.

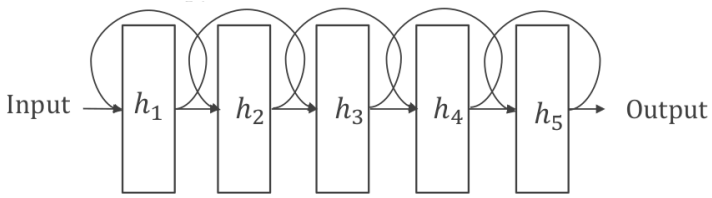

Figure 5: An RNN (Recurrent Neural Network) architecture, with *loopy* connections, such that the module's past output is its future input.

ResNet-152

ResNet-152 is a deep convolutional neural network with 152 layers. It uses a module called a 'residual block' to overcome the problem of vanishing gradients.

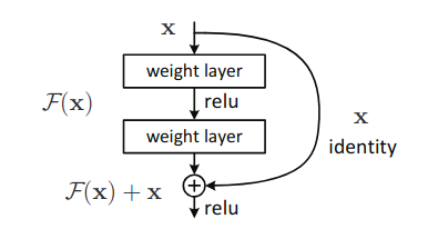

Figure 6: A residual block used in ResNet architectures.

Defined as

$$y = \mathcal{F}(x, W_i) + x, \tag{27}$$

here $\mathcal{F}$ is a 'residual function'. In essence it's a network, this network can have a single layer, or many. This allows for building the complex interconnected architecture used for image recognition and detection.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 7: Table of architectures proposed in the ResNet paper. Notice the grouping of layers into blocks/modules.

Modules in Deep Learning Frameworks
In frameworks like TensorFlow or PyTorch the module abstraction usually exists as a class.

We can define a simple module with 256 hidden units, 10 output units, using ReLU activation as follows (PyTorch)

```
import torch
from torch import nn

class MLP(nn.Module):
    def __init__(self):
        # initialize with the constructor of parent class
        super().__init__()
        self.hidden = nn.LazyLinear(256)
        self.out = nn.LazyLinear(10)

    # define the forward propagation of the model
    def forward(self, X):
        return self.out(F.relu(self.hidden(X)))
```

We usually only need to implement the forward-propagation associated with a module as these frameworks usually take care of the back-propagation.