Lecture 2 Notes

Deep Learning AI335

1 Machine Learning Paradigm

The machine learning paradigm refers to the method/approach that allows for systems to learn patterns from data and make predictions without explicit programming.

1.1 Task and Data

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:

$$f : X \rightarrow Y \tag{1}$$

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

To build a dataset to use for training we collect examples. An example is the pair (x_i, y_i) where x_i is vector containing features and y_i is it's associated label.

1.2 Model

A model is chosen to approximate f. This model will usually have some learnable parameters θ.

$$h_\theta : X \rightarrow \hat{Y} \tag{2}$$

Where $\hat{Y}$ is the predicted output space. For a single example we can write this as:

$$\hat{y}_i = h_\theta(x_i) \tag{3}$$

In the context of a neural network, each neuron performs the following computation:

$$z = \theta x + b \tag{4}$$

$$a = \sigma(z) \tag{5}$$

Where: - θ is the weight vector. - b is the bias. - σ is the activation function. - z is the linear combination of inputs. - a is the neuron's output (activation).

The network is composed of several layers of neurons. Each layer calculates:

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

$$a^{(l)} = \sigma(z^{(l)}) \tag{7}$$

Where: - $W^{(l)}$ is the weight matrix for layer l. - $b^{(l)}$ is the bias vector for layer l. - $a^{(l-1)}$ is the activation from the previous layer.

1.3 Computation

Given the set of examples D the task T and the model h_θ, The learning process or 'experience' goes as follows:

1. **Initialization**: Model h is initialized with random parameter values θ.
2. **Forward Propagation**: For each example, we compute the predicted output $\hat{y}$ using the model. *Formula (7)*
3. **Loss Computation**: We measure the loss $\mathcal{L}$ between y and $\hat{y}$
4. **Back Propagation**: We compute the gradients of the loss with respect to the model parameters θ using the chain rule.

For each layer l, we perform the following steps:

- Compute the Error Term

$$\delta^{(l)} = \begin{cases} \frac{\partial \mathcal{L}}{\partial a^{(l)}} \odot \sigma'\left(z^{(l)}\right), & \text{if } l = L \text{ (output layer)} \\ \left(W^{(l+1)}\right)^T \delta^{(l+1)} \odot \sigma'\left(z^{(l)}\right), & \text{if } l < L \end{cases} \tag{8}$$

Where: - $\delta^{(l)}$ is the error term for layer l. - $\odot$ denotes element-wise multiplication. - $\sigma'\left(z^{(l)}\right)$ is the derivative of the activation function evaluated at $z^{(l)}$.

- Compute Gradients with Respect to Parameters

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)}\left(a^{(l-1)}\right)^T \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)} \tag{10}$$

5. **Parameter Update**: We update the parameters θ using an optimization algorithm, typically gradient descent:

$$\theta := \theta - \eta \nabla_\theta \mathcal{L} \tag{11}$$

Where: - η is the learning rate. - $\nabla_\theta \mathcal{L}$ represents the gradient of the loss with respect to the parameters.

6. **Iterate**: Repeat steps 2 to 5 for multiple epochs or until a termination condition is met (e.g., acceptable error, maximum number of epochs).

2 Back-Propagation

Definition 1. Chain Rule - Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}. \tag{12}$$

Generalizing beyond the scalar case. Let $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$. If $y = g(x)$ and $z = f(y)$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i} \tag{13}$$

In vector notation, this is equivalent to

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^\top \nabla_y z \tag{14}$$

Where $\frac{\partial y}{\partial x}$ is the $n \times m$ Jacobian matrix of g.

Note 1. ∇ (*nabla*) is an operator denoting gradient. In the equation above, $\nabla_x z$ denotes the vector of partial derivatives $\frac{\partial z}{\partial x}$, representing how z changes with respect to x:

$$\nabla_x z = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \vdots \\ \frac{\partial z}{\partial x_m} \end{bmatrix}$$

Tensors
In neural networks, we usually work with tensors rather than just vectors or matrices. A tensor is an array of numbers that can have several dimensions. *Technically, vectors and matrices are tensors of rank 1 and 2.*

To apply backpropagation to a tensor, we consider its structure and apply the chain rule accordingly. Modern deep learning frameworks handle tensors efficiently without requiring explicit flattening (*converting the tensor to a vector and then reshaping it back*).

Note 2. To visualize a tensor: an image is a 2-dimensional tensor (height and width). If it has RGB color channels, it becomes a 3-dimensional tensor. A batch of these images would be a 4-dimensional tensor. *In tensor notation, the number of dimensions of a tensor is referred to as its rank.*

The chain rule in tensor notation is written as

$$\nabla_X z = \left(\frac{\partial Y}{\partial X}\right)^\top \nabla_Y z \tag{15}$$

Where X and Y are tensors, and $\frac{\partial Y}{\partial X}$ represents the Jacobian tensor.

2.1 Jacobian

Definition 2. Jacobian - For a function whose inputs and outputs are both vectors $f : \mathbb{R}^m \to \mathbb{R}^n$, the matrix containing all the partial derivatives of that function is called the Jacobian.

$$J \in \mathbb{R}^{n \times m}$$
$$J_{i,j} = \frac{\partial f_i(x)}{\partial x_j} \tag{16}$$

For a Jacobian $\frac{\partial y}{\partial x}$, the matrix describes how each component of y depends on each component of x. Each element $\frac{\partial y_j}{\partial x_i}$ tells you how the j-th element of y changes as the i-th element of x changes.

2.2 Back-Propagation as Recursive Chain Rule

In neural networks, we aim to minimize a loss function $\mathcal{L}$, which is a scalar function of the network's output. We denote the output as the activation values of the last layer, $a^{(L)}$, where L is the index of the last layer.

So we have a loss function $\mathcal{L}(y, a^{(L)})$ that depends on the network's output $a^{(L)}$, which itself depends on $a^{(L-1)}$, and so on.

The activations at each layer are computed as:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)} \tag{17}$$

$$a^{(l)} = \sigma^{(l)}\left(z^{(l)}\right) \tag{18}$$

To calculate the derivative of $\mathcal{L}$ with respect to a given layer's parameters θ_l, we apply the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial \mathcal{L}}{\partial a_L} \cdot \frac{\partial a_L}{\partial a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial a_{L-2}} \cdot \ldots \cdot \frac{\partial a_l}{\partial \theta_l} \tag{19}$$

This expression represents the recursive application of the chain rule through the layers of the network.

We can simplify this expression to two key terms:

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \left(\frac{\partial a_l}{\partial \theta_l}\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial a_l} \tag{20}$$

Here, we define the recursive relationship for the gradient with respect to activations:

$$\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial a_l}\right)^\top \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}} \tag{21}$$

Note 3. This is the simplified notation included the professors slides.

Back-Propagation Equations
For each layer l, we define the error term $\delta^{(l)}$ as:

- *Output Layer (l = L)*

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial a^{(L)}} \odot \sigma'^{(L)}\left(z^{(L)}\right) \tag{22}$$

- *Hidden Layers (l < L)*

$$\delta^{(l)} = \left(W^{(l+1)}\right)^{\top} \delta^{(l+1)} \odot \sigma'^{(l)}\left(z^{(l)}\right) \tag{23}$$

Calculating the gradients with respect to parameters

- *Weights*

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)}\left(a^{(l-1)}\right)^{\top} \tag{24}$$

- *Biases*

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)} \tag{25}$$

Note 4. The error signal $\delta^{(l)}$ represents the gradient of the loss with respect to the pre-activation $z^{(l)}$ at layer l. It captures the effect of all subsequent layers on the loss. This term propagates through the recursive process; error signals from the next layer are used to compute error signals in the current layer and so on.

Equivalence of Notations
In the professors notation the derivative of the activation function is excluded for simplicity.

$$\frac{\partial a_{l+1}}{\partial a_l} = \frac{\partial \sigma^{(l+1)}(z^{(l+1)})}{\partial a_l}$$

He also expresses the error terms in terms of this simplified gradient. We can obtain our equation by expanding.

$$\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial a^{(l)}}$$

$$\delta^{(l)} = \left(\frac{\partial a_{l+1}}{\partial a_l}\right)^{\top} \delta^{(l+1)}$$

$$\delta^{(l)} = \left(W^{(l+1)}\right)^{\top} \delta^{(l+1)} \odot \sigma'^{(l)}\left(z^{(l)}\right)$$

Which is the same as what we had in equation (23)

Another difference is in the equation for calculating the gradient with respect to parameters. This makes his (20) the 'equivalent' to our (24)

3 Modular Architecture in Neural Networks

3.1 Module

Definition 3. A module in a neural network 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.

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.

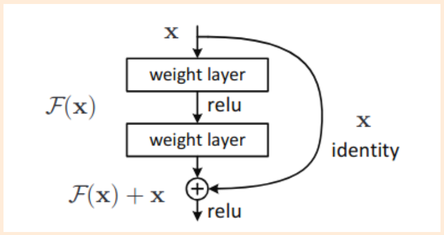
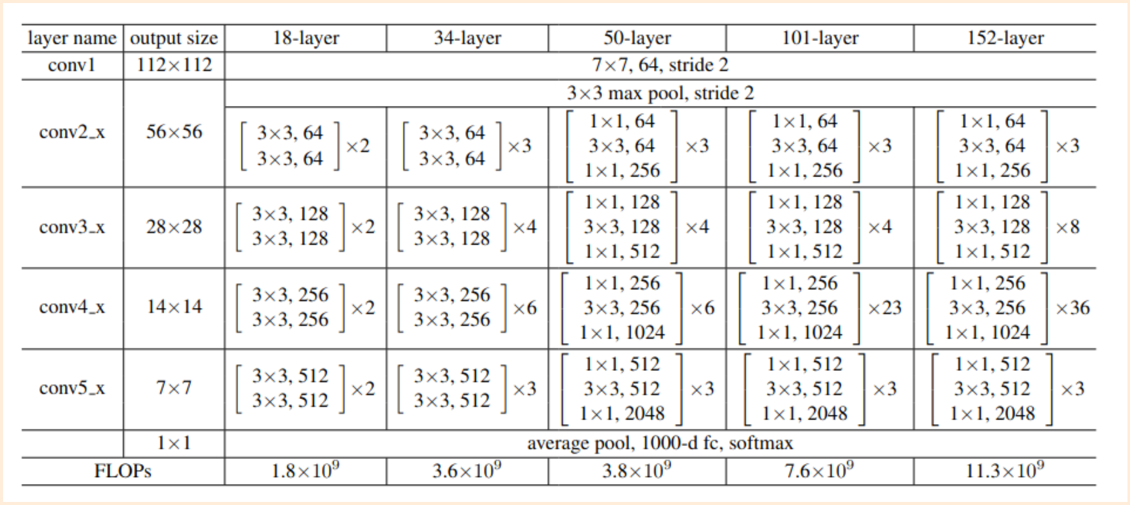

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 1: A residual block used in ResNet architectures.

Defined as

$$y = \mathcal{F}(x, W_i) + x \tag{26}$$

Here $\mathcal{F}$ is a 'residual function'. In essence it's a network, this network can have a single layer, or many. 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				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	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 2: Table of architectures proposed in the ResNet paper. Notice the grouping of layers into blocks/modules.

There are some basic requirements for valid module.

- **Paradigm Requirements**: For a module to fit into out ML paradigm it should have an input, a number of parameters associated with it's operation, and a resulting output.
- **Differentiabilty**: The module's operations must be (at least first-order) differentiable with respect to its inputs and parameters.

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

Common Modules
See slides 64-73, 83-84.