CEE 616: Probabilistic Machine Learning

M2 Deep Neural Networks: Neural Networks for Structured Data II

Jimi Oke

UMass Amherst

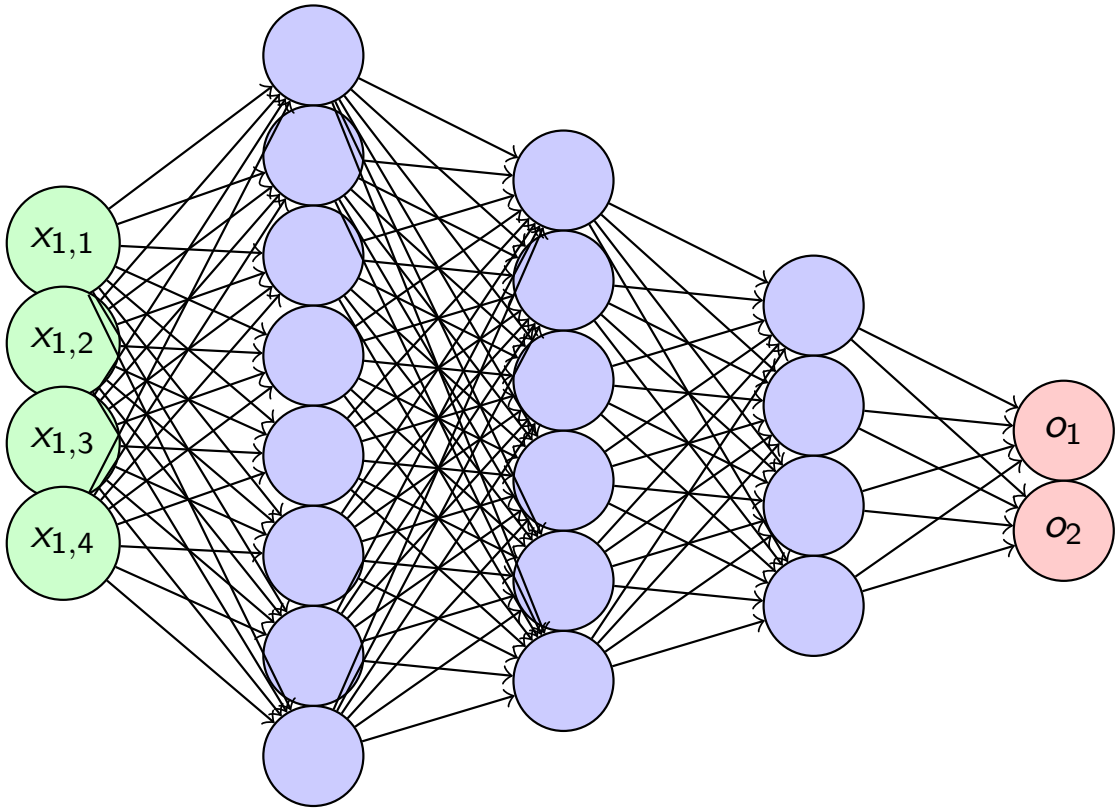
College of Engineering

Tue, Oct 21, 2025

Outline

Forward mode differentiation

Let us think of a simple feedforward with inputs $\boldsymbol{x} = \boldsymbol{x}_1 \in \mathbb{R}^{D=4}$, 3 hidden layers with $m_1 = 8$, $m_2 = 6$, and $m_3 = 4$ neurons, and outputs $\boldsymbol{o} \in \mathbb{R}^2$:



We consider each hidden unit as a function $\boldsymbol{f}_\ell(\cdot)$, where ℓ is the layer index, that maps the input $\boldsymbol{x}_\ell$ to the output of the hidden unit $\boldsymbol{x}_{\ell+1}$.

Function composition

We can then express the output of the network as a composition of functions:

$$\boldsymbol{o} = \boldsymbol{f}(\boldsymbol{x}) \tag{1}$$

where

$$\boldsymbol{f} = \boldsymbol{f}_4 \circ \boldsymbol{f}_3 \circ \boldsymbol{f}_2 \circ \boldsymbol{f}_1 \tag{2}$$

and thus

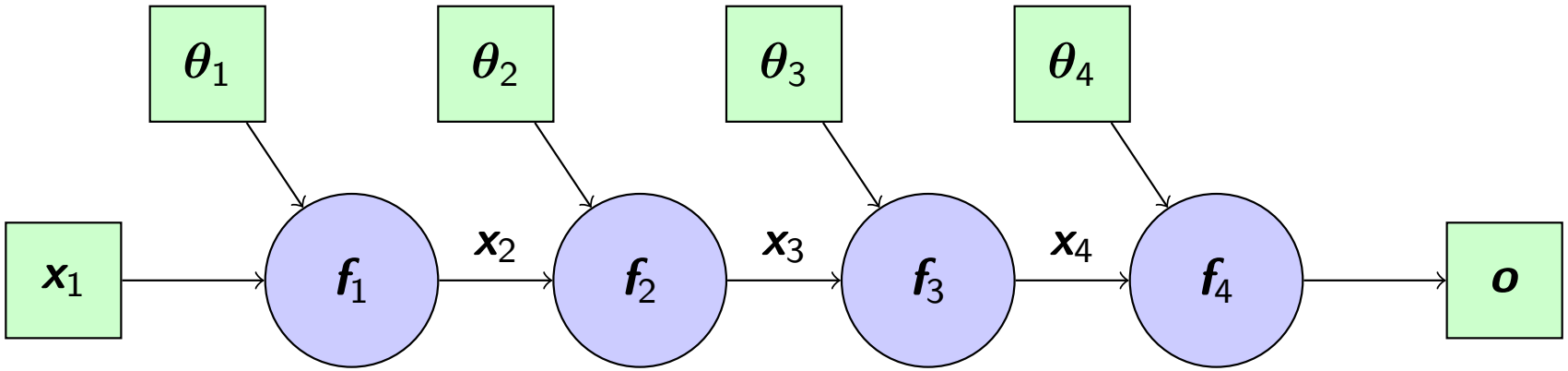
$$\boldsymbol{o} = \boldsymbol{f}_4(\boldsymbol{f}_3(\boldsymbol{f}_2(\boldsymbol{f}_1(\boldsymbol{x})))) \tag{3}$$

and

$$\begin{aligned}
\boldsymbol{x}_2 &= \boldsymbol{f}_1(\boldsymbol{x}_1), \quad \boldsymbol{f}_1 : \mathbb{R}^4 \to \mathbb{R}^8 \\
\boldsymbol{x}_3 &= \boldsymbol{f}_2(\boldsymbol{x}_2), \quad \boldsymbol{f}_2 : \mathbb{R}^8 \to \mathbb{R}^6 \\
\boldsymbol{x}_4 &= \boldsymbol{f}_3(\boldsymbol{x}_3), \quad \boldsymbol{f}_3 : \mathbb{R}^6 \to \mathbb{R}^4 \\
\boldsymbol{o} &= \boldsymbol{f}_4(\boldsymbol{x}_4), \quad \boldsymbol{f}_4 : \mathbb{R}^4 \to \mathbb{R}^2
\end{aligned}$$

Computational graph

We can further visualize the FFNN as a computational graph:



where θ_ℓ are the parameters (weights and biases) of layer ℓ.

Backpropagation

To compute the gradient, we need to find:

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{f}_4(\boldsymbol{x}_4)}{\partial \boldsymbol{x}_4} \cdot \frac{\partial \boldsymbol{f}_3(\boldsymbol{x}_3)}{\partial \boldsymbol{x}_3} \cdot \frac{\partial \boldsymbol{f}_2(\boldsymbol{x}_2)}{\partial \boldsymbol{x}_2} \cdot \frac{\partial \boldsymbol{f}_1(\boldsymbol{x}_1)}{\partial \boldsymbol{x}_1} \tag{4}$$

We define the Jacobian matrix of each layer as:

$$\boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x}_\ell) = \frac{\partial \boldsymbol{f}_\ell(\boldsymbol{x}_\ell)}{\partial \boldsymbol{x}_\ell} = \begin{pmatrix} \frac{\partial f_{\ell,1}(\boldsymbol{x}_\ell)}{\partial x_{\ell,1}} & \frac{\partial f_{\ell,1}(\boldsymbol{x}_\ell)}{\partial x_{\ell,2}} & \cdots & \frac{\partial f_{\ell,1}(\boldsymbol{x}_\ell)}{\partial x_{\ell,D}} \\ \frac{\partial f_{\ell,2}(\boldsymbol{x}_\ell)}{\partial x_{\ell,1}} & \frac{\partial f_{\ell,2}(\boldsymbol{x}_\ell)}{\partial x_{\ell,2}} & \cdots & \frac{\partial f_{\ell,2}(\boldsymbol{x}_\ell)}{\partial x_{\ell,D}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{\ell,M}(\boldsymbol{x}_\ell)}{\partial x_{\ell,1}} & \frac{\partial f_{\ell,M}(\boldsymbol{x}_\ell)}{\partial x_{\ell,2}} & \cdots & \frac{\partial f_{\ell,M}(\boldsymbol{x}_\ell)}{\partial x_{\ell,D}} \end{pmatrix} \tag{5}$$

Thus, the gradient of the output with respect to the input is given by:

$$\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{x}} = \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x}_4) \cdot \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x}_3) \cdot \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x}_2) \cdot \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x}_1) \tag{6}$$

Jacobian matrix (2/2)

We can write the Jacobian as:

$$\boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x}) = \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{pmatrix} \nabla f_1(\boldsymbol{x})^\top \\ \nabla f_2(\boldsymbol{x})^\top \\ \vdots \\ \nabla f_M(\boldsymbol{x})^\top \end{pmatrix} = \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}_1}, \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}_2}, \ldots, \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}_D} \right) \tag{7}$$

- i'th row: $\nabla f_i(\boldsymbol{x})^\top \in \mathbb{R}^{1 \times D}$ gradient of the i'th output w.r.t. all inputs
- j'th column: $\frac{\partial \boldsymbol{f}}{\partial x_j} \in \mathbb{R}^M$ gradient of all outputs w.r.t. the j'th input

Thus:

$$\nabla \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{x})^\top = (\nabla f_1(\boldsymbol{x}), \nabla f_2(\boldsymbol{x}), \ldots, \nabla f_M(\boldsymbol{x})) \tag{8}$$

Vector Jacobian product

The vector-Jacobian product (VJP) is the left-multiplication of the Jacobian matrix $\boldsymbol{J_f}(\boldsymbol{x})$ by a vector $\boldsymbol{u}$, which results in a row vector:

$$\boldsymbol{u}^\top \boldsymbol{J_f}(\boldsymbol{x}) \in \mathbb{R}^{1 \times D} \tag{9}$$

- If $\boldsymbol{u} \in \mathbb{R}^M$ is a one-hot vector with 1 at index i and 0 elsewhere, e.g.

$$\boldsymbol{u} = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} = \boldsymbol{e}_i \tag{10}$$

then the VJP $\boldsymbol{e}_i^\top \boldsymbol{J_f}(\boldsymbol{x})$ extracts the i'th row from $\boldsymbol{J_f}(\boldsymbol{x})$, which is the gradient of the i'th output with respect to all inputs, $\nabla f_i(\boldsymbol{x})^\top$.

Jacobian-vector product

The Jacobian-vector product (JVP) is the right-multiplication of the Jacobian matrix $\boldsymbol{J_f}(\boldsymbol{x})$ by a vector $\boldsymbol{v}$, which results in a column vector:

$$\boldsymbol{J_f}(\boldsymbol{x})\boldsymbol{v} \in \mathbb{R}^M \tag{11}$$

- If $\boldsymbol{v} \in \mathbb{R}^D$ is a one-hot vector with 1 at index j and 0 elsewhere, e.g.

$$\boldsymbol{v} = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} = \boldsymbol{e}_j \tag{12}$$

then the JVP $\boldsymbol{J_f}(\boldsymbol{x})\boldsymbol{e}_j$ extracts the j'th column from $\boldsymbol{J_f}(\boldsymbol{x})$, which is the gradient of all outputs with respect to the j'th input, $\frac{\partial \boldsymbol{f}}{\partial x_j}$.

Computing the Jacobian

- The Jacobian can be computed by performing either M JVPs (one per input dimension) or D VJPs (one per output dimension)
- If $D < M$, it is more efficient to compute the Jacobian via JVPs for each column j:

$$\boldsymbol{J_f}(\boldsymbol{x})(\boldsymbol{v}) = \underbrace{\boldsymbol{J_f}(\boldsymbol{x})_4}_{M \times M_3} \underbrace{\boldsymbol{J_f}(\boldsymbol{x})_3}_{M_3 \times M_2} \underbrace{\boldsymbol{J_f}(\boldsymbol{x})_2}_{M_2 \times M_1} \underbrace{\boldsymbol{J_f}(\boldsymbol{x})_1}_{M_1 \times D} \underbrace{\boldsymbol{v}}_{D \times 1} \tag{13}$$

  in a right-to-left fashion (**forward mode differentiation**)
- If $M < D$, it is more efficient to compute the Jacobian via VJPs for each row i:

$$\boldsymbol{u}^\top \boldsymbol{J_f}(\boldsymbol{x}) = \underbrace{\boldsymbol{u}^\top}_{1 \times M} \underbrace{\boldsymbol{J_f}(\boldsymbol{x})_4}_{M \times M_3} \underbrace{\boldsymbol{J_f}(\boldsymbol{x})_3}_{M_3 \times M_2} \underbrace{\boldsymbol{J_f}(\boldsymbol{x})_2}_{M_2 \times M_1} \underbrace{\boldsymbol{J_f}(\boldsymbol{x})_1}_{M_1 \times D} \tag{14}$$

  in a left-to-right fashion (**reverse mode differentiation**)
- In neural networks, the number of outputs M is often much smaller than the number of inputs D
- Thus, backpropagation typically employs VJPs to compute the gradients efficiently

Example: single hidden layer MLP

Consider a single hidden layer MLP with an ℓ_2 loss for regression (scalar output):

$$\mathcal{L}((\boldsymbol{x}, \boldsymbol{y}), \boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{y} - \boldsymbol{W}_2\varphi(\boldsymbol{W}_1\boldsymbol{x})||_2^2 \tag{15}$$

We can represent the network as follows:

$$\begin{aligned}
\mathcal{L} &= \boldsymbol{f}_4 \circ \boldsymbol{f}_3 \circ \boldsymbol{f}_2 \circ \boldsymbol{f}_1 && (16)\\
\boldsymbol{x}_2 &= \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{\theta}_1) = \boldsymbol{W}_1\boldsymbol{x} && (17)\\
\boldsymbol{x}_3 &= \boldsymbol{f}_2(\boldsymbol{x}_2, \boldsymbol{\theta}_2) = \varphi(\boldsymbol{x}_2) && (18)\\
\boldsymbol{x}_4 &= \boldsymbol{f}_3(\boldsymbol{x}_3, \boldsymbol{\theta}_3) = \boldsymbol{W}_2\boldsymbol{x}_3 && (19)\\
\mathcal{L} &= \boldsymbol{f}_4(\boldsymbol{x}_4, \boldsymbol{y}) = \frac{1}{2}||\boldsymbol{y} - \boldsymbol{x}_4||_2^2 && (20)
\end{aligned}$$

and $\mathcal{L} : \mathbb{R}^D \to \mathbb{R}$.

Example: single hidden layer MLP (contd.)

Using the chain rule, We can express the gradient of the loss wrt the parameters in each layer as:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_3} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_4}\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{\theta}_3} \tag{21}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_2} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_4}\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{x}_3}\frac{\partial \boldsymbol{x}_3}{\partial \boldsymbol{\theta}_2} \tag{22}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_1} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_4}\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{x}_3}\frac{\partial \boldsymbol{x}_3}{\partial \boldsymbol{x}_2}\frac{\partial \boldsymbol{x}_2}{\partial \boldsymbol{\theta}_1} \tag{23}$$

Note:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k} = (\nabla_{\boldsymbol{\theta}_k} \mathcal{L})^\top \tag{24}$$

is a d_k-dimensional gradient row vector, where d_k is the number of parameters in layer k and can be computed recursively as VJPs:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k} = \boldsymbol{u}_{k+1}^\top \frac{\partial \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)}{\partial \boldsymbol{\theta}_k} \tag{25}$$

where $\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)$, $\boldsymbol{u}_k^\top = \boldsymbol{u}_{k+1}^\top \frac{\partial \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{\theta}_k)}{\partial \boldsymbol{x}_k}$ and $\boldsymbol{u}_{K+1} = 1$.

Example: single hidden layer MLP (contd.)

We then need to specify the Jacobians for each layer:

- Output layer:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_4} = \boldsymbol{u}_4^\top = \frac{\partial \frac{1}{2}||\boldsymbol{y} - \boldsymbol{x}_4||_2^2}{\partial \boldsymbol{x}_4} = (\boldsymbol{x}_4 - \boldsymbol{y})^\top \tag{26}$$

- Linear layer:

$$\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{x}_3} = \boldsymbol{J}_{\boldsymbol{f}_3}(\boldsymbol{x}_3) = \frac{\partial \boldsymbol{W}_2 \boldsymbol{x}_3}{\partial \boldsymbol{x}_3} = \boldsymbol{W}_2 \tag{27}$$

$$\frac{\partial \boldsymbol{x}_4}{\partial \boldsymbol{W}_2} = \frac{\partial \boldsymbol{W}_2 \boldsymbol{x}_3}{\partial \boldsymbol{W}_2} = \boldsymbol{x}_3^\top \tag{28}$$

Batch learning

The gradient descent method was described without referencing the observations.

In reality, the forward and backward passes are performed for each observation in the training set, with parameter updates obtained by **averaging** the gradients:

$$\boldsymbol{\theta}_{\ell,t+1} = \boldsymbol{\theta}_{\ell,t} - \rho \frac{1}{N} \sum_{n=1}^{N} \frac{\partial \mathcal{L}_n(\boldsymbol{\theta}_{\ell,t})}{\partial \boldsymbol{\theta}_\ell} \tag{29}$$

This approach is called **batch learning**.

- One sweep through all the training observations is called an *epoch*
- In batch learning, only one update results from a training epoch
- Thus, several epochs are required for convergence

Stochastic gradient descent

In standard gradient descent (batch learning, the updates are performed only after the gradient is computed for *all* training observations.

The stochastic gradient descent approach approximates the gradient in each iteration using a randomly selected observation n:

$$\boldsymbol{\theta}_{\ell,t+1} = \boldsymbol{\theta}_{\ell,t} - \rho \frac{\partial \mathcal{L}_n(\boldsymbol{\theta}_{\ell,t})}{\partial \boldsymbol{\theta}_\ell} \tag{30}$$

$$\tag{31}$$

- Each iteration over observation n results in a weight/bias update
- Thus, one sweep through the entire training set (an epoch) produces N updates

This procedure is also known as **online learning**

Mini-batch learning

To introduce stability, we can compute weight updates over a *subset* of training observations

1. Randomly sample a mini-batch $\mathcal{B}_t$ of B samples (this means that $|\mathcal{B}_t| = B$), where $B \ll N$.
2. Perform a forward and backward pass through each mini-batch to update the weights:

$$\boldsymbol{\theta}_{\ell,t+1} = \boldsymbol{\theta}_{\ell,t} - \rho \frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \frac{\partial \mathcal{L}_m(\boldsymbol{\theta}_{\ell,t})}{\partial \boldsymbol{\theta}_\ell} \tag{32}$$

   Thus, for each iteration, average the gradient over a *mini-batch* of B randomly selected observations
3. Repeat step 2 until convergence

Considerations

- Online learning is more efficient for very large datasets
- In practice, mini-batch learning is employed, as batch sizes (usually, $M = 16$ or $M = 32$) can be chosen to take advantage of parallel computing architectures
- An **adpative** learning rate ρ can guarantee convergence, e.g. $\rho_r = \frac{1}{r}$
- Input **standardization** (mean zero, SD 1) is recommended for consistent weight initialization and regularization
- Weights/biases are **initialized** to *small* values near 0 for better performance
- Cost function C is nonlinear and nonconvex; other optimization approaches (e.g. conjugate gradient) can provide faster convergence compared to stochastic gradient descent

Regularization

DNNs are prone to overfitting. To migitate this, we can regularize them by:

- Early stopping: terminating training if objective does not improve after a specified number of epochs (patience)
- Weight decay:

$$C \leftarrow C + \lambda J = C + \lambda \left(\sum w^2 + \sum b^2 \right) \tag{33}$$

  where λ is tuning parameter estimated via cross-validation
- Model compression via ℓ_1 regularization of weights (sparse DNNs)
- Dropout: randomly switching off connections from each neuraon with probability p

Regression MLP architecture

Typical hyperparameter values are:

Hyperparameter	**Value**
# input neurons	1 per input feature
# hidden layers	Usually 1 – 5
# neurons per hidden layer	Usually 10 – 100
# output neurons	1 per prediction dimension
hidden layer activation	ReLU
output activation	None (if unbounded)
loss function	MSE or MAE/Huber

Classification MLP architecture

- For classifcation, input and hidden layers are chosen in similar fashion to the regression case
- However, the number of output neurons is given by the name of classes/labels
- The output layer activation is typically the softmax function:

$$o_c = \mathcal{S}(\boldsymbol{a}_c) = \frac{e^{\boldsymbol{a}_c}}{\sum_{c'=1}^{C} e^{\boldsymbol{a}_{c'}}} \tag{34}$$

  where $\boldsymbol{a}_c$ is the unnormalized log probability of each class c
- The loss function is taken as the **categorical cross-entropy**:

$$\mathcal{L} = -\sum_{c=1}^{C} y_c \log p_c = -\sum_{c=1}^{C} y_c \log(\mathcal{S}(\boldsymbol{a}_c)) \tag{35}$$

Other types of neural networks

The standard ANN architecture (MLP) we have studied is also called the feed-forward network.

Other architectures have been shown to give better performance for various applications:

- Recurrent neural networks (RNNs): time-series forecasting
- Convolutional neural networks (CNNs): image classification
- Long short-term memory networks (LSTMs): time-series, pattern identification, etc.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As the
unprocessed data that should have been added to the final page this e
has been added to receive it.
If you rerun the document (without altering it) this surplus page will g
because LaTeX now knows how many pages to expect for this documen