Deep Learning Lecture Notes

Henry Baker

2024

Contents

Chapter 1

Deep Learning in Public Policy

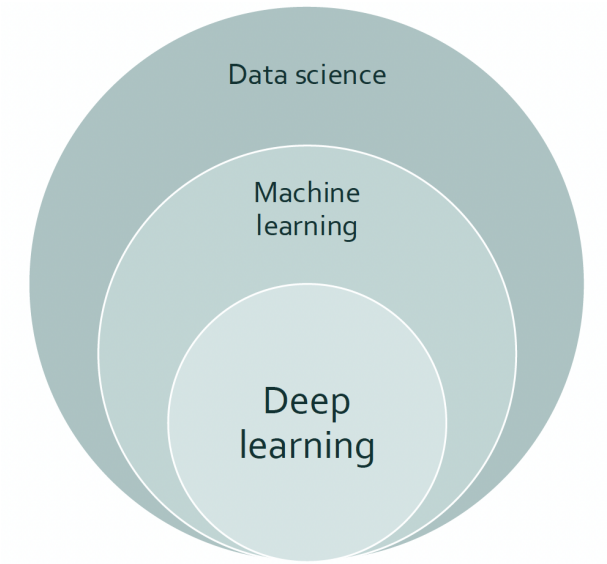


Figure 1.1: Enter Caption

In DL, we are interested in finding the relationship $Y = f(X) + \epsilon$.

Statistics

- Descriptive
- Inferential

Machine Learning

- Predictive
- Leveraging CS techniques
- Large datasets

Supervised - for each observation x_i there is an associated response measurement y_i.

Unsupervised - finding patters in data w/o a response y_i. - eg clustering; topic modeling; GPT & all generative models.

Reinforcement learning - Finding the optimal policy based on states and actions that maximizes a reward; games, robotics, engineering control

---

1.1 ML vs DL

ML:

- SVMs
- Random Forests
- ...
- NNs

DL:

- Deep/multlayered NNs

1.2 Universal Approximation Theorem

Posits a single-layer neural network as the universal approximator.

Universal Approximation Theorem (Hornik, 1991):

"A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units"

- This applies for sigmoid, tanh and many other activation functions.
- However, this does not mean that there is learning algorithm that can find the necessary parameter values.

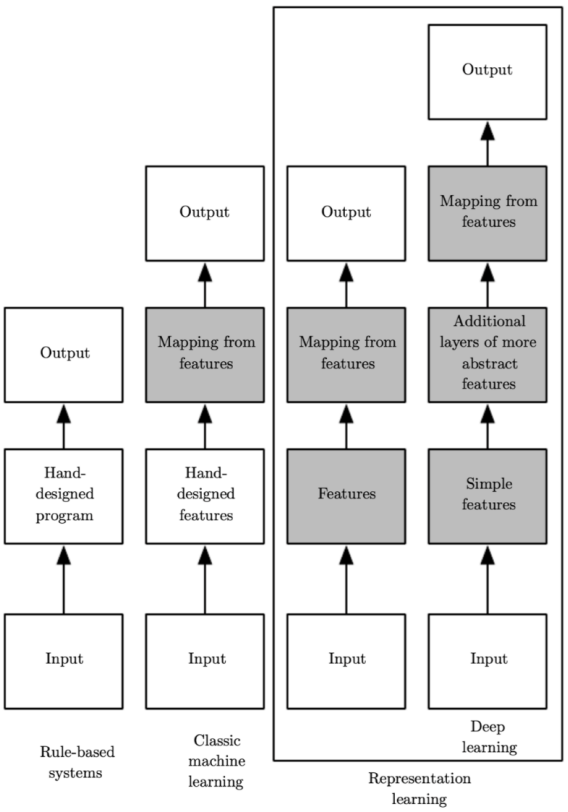
In plain terms: *in theory*, single hidden layer neural networks are very good at approximating any function... but they may not learn well *in practice*. In contrast, deep learning models, perform well in practice.

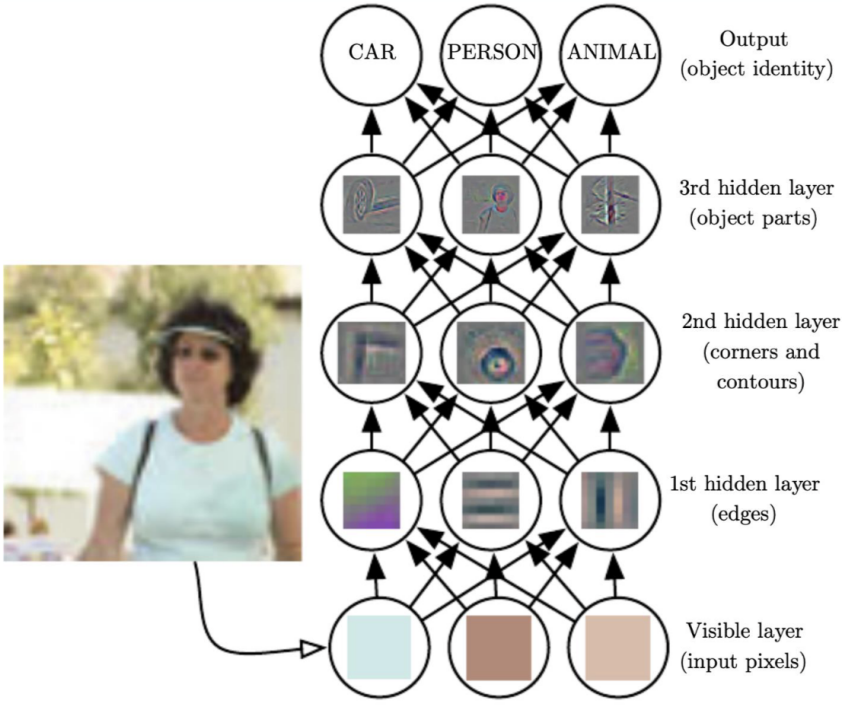
The question is **why** is DL so successful? Empirically it is the case, but we lack mathematical theory for why.

1.3 What is DL

- Traditional ML often relies on **feature extraction and engineering on the raw data** input (scaling, clustering, categorical encoding, domain knowledge-based features...)

- DL handles that within the model itself; it introduces representations of high-dimensional data by expressing them in terms of simpler representations **(DL does representation learning / feature learning)**.
- Still, not all feature engineering is useless in DL (e.g. tokenisation etc in NLP ???)





More flexibility in representing features in a hierarchical way

Some modern deep learning:

- Computer vision w/ convolutional layers
- Sequence modeling w/ feedback connections (LTSM)
- Language modeling w/ attention

... the lecture goes on to cover i) basics of DL in policy context, ii) policy for DL.

Chapter 2

Deep Neural Networks I

Standard definition of a linear transformation in mathematics:

$$y = Ax$$

So in DL: when we have x_i:

$$y = Wx + b$$

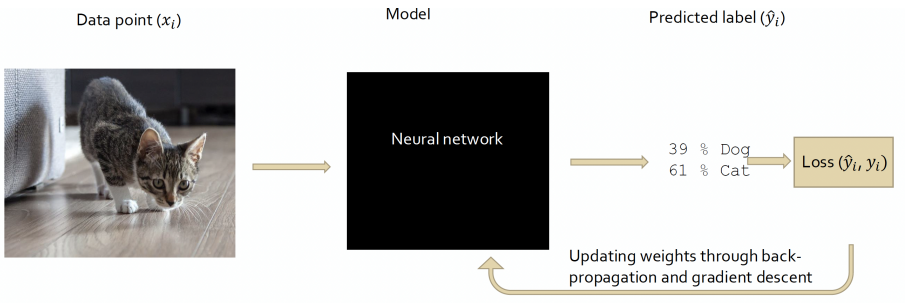
When we have X:

$$Y = XW + b$$

2.1 Overview

- Overview of how NNs 'learn'
- Structure of a single-layer NN
- Gradient descent
- Training the network and back-propagation

2.2 High-level overview: How a neural network learns



1. **Feed-forward** - takes an input example, feeds it into a NN, predicts a vector that gives % of class label (this assumes some preset configuration of the model), then does an initial attempt at a classification.
2. **Computing Loss & Back-propagation** - use info about the loss (the gradient) to change our model.

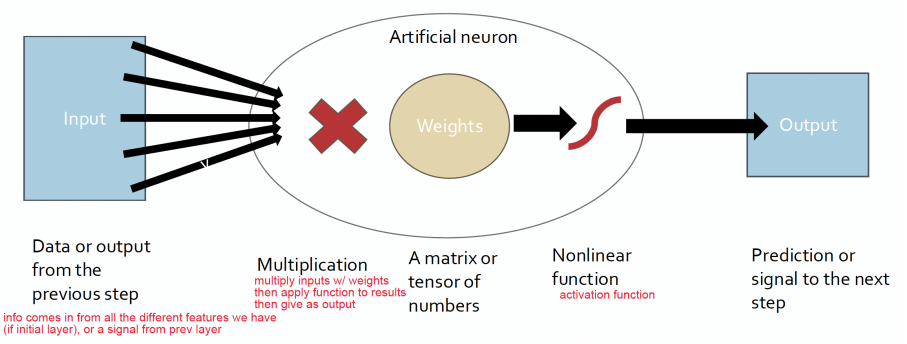
(a) Make a prediction,

(b) See how wrong we are,

(c) Use that info to move in the right direction to be more right.

3. **Hyper-parameter tuning on validation set**
4. **Reporting performance on test set**

2.2.1 Artificial Neurons (Hidden units)



- **Input Activation Matrix** $X \in \mathbb{R}^{n \times d}$

  As a batch (a matrix):

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{bmatrix}$$

Treated as a single unit x (a column vector):

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_d \end{bmatrix}$$

- **Multiplied by Weight Matrix** $W^{[1]} \in \mathbb{R}^{d \times h}$

  – h is the number of neurons in the current layer.

  – d is the number of neurons in the previous layer (number of input features).

  Contains $d \times h$ weights:

$$\mathbf{W}^{[1]} = \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & \cdots & w_{1h}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & \cdots & w_{2h}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1}^{[1]} & w_{d2}^{[1]} & \cdots & w_{dh}^{[1]} \end{bmatrix}$$

Treated as a single hidden unit's activations (a row column vector):

$$\mathbf{w} = \begin{bmatrix} w_1 & w_2 & \cdots & w_d \end{bmatrix}$$

The weight matrix $W_i^{[1]}$ transforms input data from a dimension of d (number of input features) to a dimension of H (number of hidden units or neurons in the subsequent layer).

- Each column of $W^{[1]}$ typically represents the weights connecting all input features to a single hidden unit. *This is what W^T does when we treat it as a lone vector.*
- Matrix multiplication $W^{[1]}$ results in a transformation from the input space to the hidden layer space.

(Once the neuron output activation function has also been applied, this transformation results in the hidden layer activation matrix $H \in \mathbb{R}^{n \times h}$, calculated as:

$$\mathbf{H}_{n,h} = \sigma\Big(\mathbf{X}_{n,d}\,\mathbf{W}_{d,h}^{[1]} + \mathbf{1}_n\,\mathbf{b}_h^{[1]T}\Big)$$

NB: The row dimension n never changes as you go deeper — each input sample carries forward through the network. What changes is only the column dimension (the number o features/activations), which depends on how many hidden units the layer has.

Hidden Layer Activation Matrix $\mathbf{H} \in \mathbb{R}^{n \times h}$

$$\mathbf{H} = \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1h} \\ h_{21} & h_{22} & \cdots & h_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nh} \end{bmatrix}$$

Each element h_{ij} of $\mathbf{H}$ represents the activation of the j-th hidden unit for the i-th input sample. The hidden layer activation matrix $\mathbf{H}$ can be interpreted as follows:

- **Rows** correspond to the activations of the hidden units for each input sample. Each row in $\mathbf{H}$ will represent how the network transforms that specific input based on the weights and biases of the hidden layer.
  - Example: If we have 10 input samples and 5 hidden units, the matrix will have 10 rows. The first row would tell us the activations of the hidden units for the first input sample, the second row for the second input, and so on.
- **Columns** correspond to the activations of a particular hidden unit across all input samples. The values in the column show how each input sample contributes to the activation of that particular hidden unit.
  - Example: If the second column corresponds to the activations of the second hidden unit, then the values in that column show how the second hidden unit reacts to all the input samples in the batch.

... jumped ahead a bit there, returning back to slides.

- **Neuron *input* activation for data** $X \in \mathbb{R}^{n \times d}$

$$\begin{aligned} a(X) &= b + \sum_j^d w_j x_j \\ &= b + W^T X \end{aligned}$$

  - X - input activation matrix $(n \times d)$, but each observation in X is treated as a single vector $\mathbb{R} \in$
  - W^T - vector of length (d)
  - b - bias term (scalar)

NB - Bias Term: provides an **additional degree of freedom**; it allows unit's activation function to be shifted to the **left or right**, which can be crucial for the learning process. It helps the model fit the data better by providing additional flexibility.

NB - Matrix Multiplication Order & Dimensions:

CHECK

Vector Case: here we have $W^T X$

  - $X \in \mathbb{R}^d$ - column vector
  - $W \in \mathbb{R}^d$ - column vector
  - $W^T \in \mathbb{R}^{1 \times d}$ - row vector

$$W^T_{(1 \times d)} X_{(d \times 1)}$$

  - a i.e. $(W^T X)$ = scalar

Matrix Case: here we have XW

  - $X \in \mathbb{R}^{n \times d}$ - input matrix (with n samples, each having d features)
  - $W \in \mathbb{R}^{d \times h}$ - weight matrix (mapping from d features to h hidden units)

$$X_{(n \times d)} W_{(d \times h)} = (XW) \in \mathbb{R}^{n \times h}$$

  - $a(X)$, i.e., XW is now $\mathbb{R}^{n \times h}$ - result is a matrix of size $n \times h$

Matrix multiplication $W^{[1]}$ results in a transformation from the input space to the hidden layer space.

- **Neuron *output* activation**:

$$\begin{aligned} h(X) &= g(a(X)) \\ &= g\left(b + \sum_j^d w_j x_j\right) \end{aligned}$$

– $g(\cdot)$ - activation function, applied piece wise to input activation.

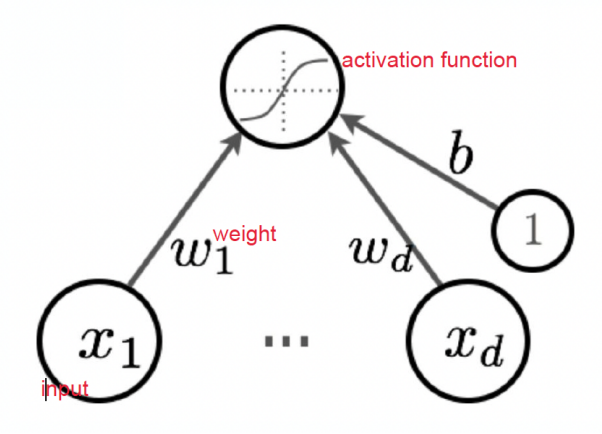


Figure 2.1: hidden unit's connection to input activation

- Non-linear activation function allows the NN to be expressive (w/o which the layers collapse and it becomes one big linear regression equation)

Looking at hidden unit i: i think normal notation is hidden units j, inputs i, but I'm following lecture slides here.

$$\begin{aligned} h_i(X) &= g(a_i(X)) \\ &= g\left(b_i + \sum_j^d w_{ij}^{[1]} x_j\right) \end{aligned}$$

2.2.2 Output layer (in a single-layered NN)

Again consists of a linear combination of the hidden units, with an activation function $o(\cdot)$:

$$f(X) = o\left(b^{[2]} + \sum_{i=1}^{H} w_i^{[2]} x_i\right)$$

Generic $l(-1?)$ layers:

$$f(X) = o\left(b^{[l]} + \sum_{i=1}^{H} w_i^{[l]} x_i\right)$$

Where

- $\sum_{i=1}^{H} \ldots$ - **weighted sum of inputs:**
  - H represents the number of hidden units in the preceding layer
  - $w_i^{[l]}$ are the weights connecting the ith hidden unit from the layer $l-1$ to the (single) output neuron
  - x_i are the activations (outputs) from the hidden units in layer $l-1$

- $o(\cdot)$ - activation function

- $b^{[l]}$ - bias term for penultimate layer

2.2.3 Single-layer NNs

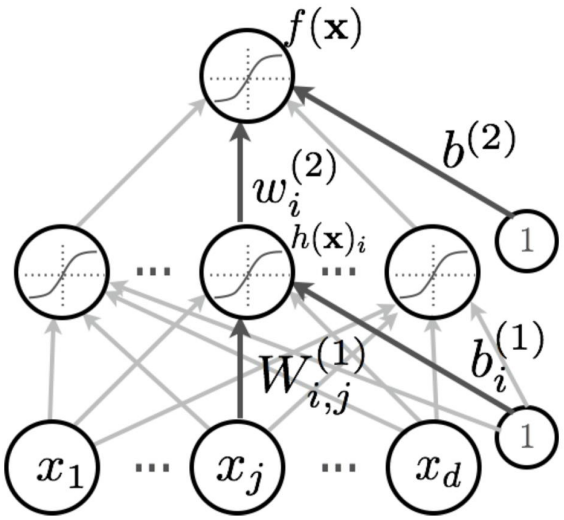


Figure 2.2: NB: $W^{[1]}$ is a matrix; $w^{(2)}$ is a vector!

- i - index for hidden neurons

- j - index for input features

- **Weights:** $w_{ij}^{[l]}$ connects the j-th neuron from layer $l-1$ to the i-th neuron in layer l. NB!!!

- **Biases:** $b_i^{[l]}$ is the bias for the i-th neuron in layer l.

$\mathbf{H} = \mathbf{XW}$:

$$X_{(n,d)} \times W_{(d,h)} = H_{(n \times h)}$$

$$\begin{pmatrix} \mathbf{x_{11}} & \mathbf{x_{12}} & \cdots & \mathbf{x_{1d}} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} \times \begin{pmatrix} \mathbf{w_{11}} & w_{12} & \cdots & w_{1h} \\ \mathbf{x_{21}} & w_{22} & \cdots & w_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x_{d1}} & w_{d2} & \cdots & w_{dh} \end{pmatrix} = \begin{pmatrix} \mathbf{h_{11}} & h_{12} & \cdots & h_{1h} \\ h_{21} & h_{22} & \cdots & h_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nh} \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{x_{11}} & \mathbf{x_{12}} & \cdots & \mathbf{x_{1d}} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} \times \begin{pmatrix} w_{11} & \mathbf{w_{12}} & \cdots & w_{1h} \\ w_{21} & \mathbf{x_{22}} & \cdots & w_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & \mathbf{x_{d2}} & \cdots & w_{dh} \end{pmatrix} = \begin{pmatrix} h_{11} & \mathbf{h_{12}} & \cdots & h_{1h} \\ h_{21} & h_{22} & \cdots & h_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nh} \end{pmatrix}$$

$$\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ \mathbf{x_{21}} & \mathbf{x_{22}} & \cdots & \mathbf{x_{2d}} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} \times \begin{pmatrix} \mathbf{w_{11}} & w_{12} & \cdots & w_{1h} \\ \mathbf{w_{21}} & w_{22} & \cdots & w_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w_{d1}} & w_{d2} & \cdots & w_{dh} \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & \cdots & h_{1h} \\ \mathbf{h_{21}} & h_{22} & \cdots & h_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nh} \end{pmatrix}$$

$$\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ \mathbf{x_{21}} & \mathbf{x_{22}} & \cdots & \mathbf{x_{2d}} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} \times \begin{pmatrix} w_{11} & \mathbf{w_{12}} & \cdots & w_{1h} \\ w_{21} & \mathbf{w_{22}} & \cdots & w_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & \mathbf{w_{d2}} & \cdots & w_{dh} \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & \cdots & h_{1h} \\ h_{21} & \mathbf{h_{22}} & \cdots & h_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \cdots & h_{nh} \end{pmatrix}$$

Each element of h_{ij} of H is computed as the dot product of a row of X and a column of W:

$$h_{ij} = \sum_{k=1}^{d} X_{ik} W_{kj}$$

NB how $w_{ij}^{[l]}$ connects

- i: layer l neuron i
- j: layer $l-1$, neuron j

Treating X as a single row observation x_i: a vector $\in \mathbb{R}^d$:

$$x_{i_{(1,d)}} \times W_{(d,h)} = H_{i_{(1,h)}}$$

NB This highlights how a single observation x_i is transformed by the matrix from a d dimensional (transposed column) vector into an h dimensional vector (column) in the hidden layer's H matrix.

This can be expressed as

$$h_{ik} = \sum_{j=1}^{d} x_{ij} w_{jk}$$

These row vectors are effectively stacked into an H matrix.

$$\begin{pmatrix} \mathbf{x_{i1}} & \mathbf{x_{i2}} & \dots & \mathbf{x_{id}} \end{pmatrix} \times \begin{pmatrix} \mathbf{w_{11}} & w_{12} & \dots & w_{1h} \\ \mathbf{w_{21}} & w_{22} & \dots & w_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w_{d1}} & w_{d2} & \dots & w_{dh} \end{pmatrix} = \begin{pmatrix} \mathbf{h_{i1}} & h_{i2} & \dots & h_{ih} \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{x_{i1}} & \mathbf{x_{i2}} & \dots & \mathbf{x_{id}} \end{pmatrix} \times \begin{pmatrix} w_{11} & \mathbf{w_{12}} & \dots & w_{1h} \\ w_{21} & \mathbf{w_{22}} & \dots & w_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & \mathbf{w_{d2}} & \dots & w_{dh} \end{pmatrix} = \begin{pmatrix} h_{i1} & \mathbf{h_{i2}} \dots & h_{ih} \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{x_{i1}} & \mathbf{x_{i2}} & \dots & \mathbf{x_{id}} \end{pmatrix} \times \begin{pmatrix} w_{11} & w_{12} & \mathbf{w_{13}} & \dots \\ w_{21} & w_{22} & \mathbf{w_{23}} & \dots \\ \vdots & \vdots & \vdots & \ddots \\ w_{d1} & w_{d2} & \mathbf{w_{d3}} & \dots \end{pmatrix} = \begin{pmatrix} h_{i1} & h_{i2} & \mathbf{h_{i3}} & h_{ih} \end{pmatrix}$$

$$\begin{pmatrix} \mathbf{x_{i1}} & \mathbf{x_{i2}} & \dots & \mathbf{x_{id}} \end{pmatrix} \times \begin{pmatrix} w_{11} & w_{12} & \dots & \mathbf{w_{1h}} \\ w_{21} & w_{22} & \dots & \mathbf{w_{2h}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & w_{d2} & \dots & \mathbf{w_{dh}} \end{pmatrix} = \begin{pmatrix} h_{i1} & h_{i2} & \dots \mathbf{h_{ih}} \end{pmatrix}$$

Alternatively, if we want to emphasize the contribution to a specific hidden unit h_{ik} we focus on a particular column of W

$$\begin{pmatrix} \mathbf{x_{i1}} & \mathbf{x_{i2}} & \dots & \mathbf{x_{id}} \end{pmatrix} \times \begin{pmatrix} \mathbf{w_{1k}} \\ \mathbf{w_{2k}} \\ \vdots \\ \mathbf{w_{dk}} \end{pmatrix} = \begin{pmatrix} \mathbf{h_{ik}} \end{pmatrix}$$

This highlights that just the x_i observation contributes only one value (h_{ik}) to the kth hidden unit vector

For final layer in multiclass classification, where $\mathbb{W}^{[2]}$ is a matrix to produce an output matrix Z consisting of stacked vectors $z_i \in \mathbb{R}^c$ for each observation

$$H_{(n,h)} \times W^{[2]}_{(h,c)} = Z_{n,c}$$

This shows how it collapses it down to a vector output of c dimensions for each x_i observation. These row vectors are stacked into a matrix Z of $n \times c$.

$$H_{(n,h)} \times W^{[2]}_{(h,c)} = Z_{(n,c)}$$

$$\begin{pmatrix} h_{11} & h_{12} & \dots & h_{1h} \\ \mathbf{h_{21}} & \mathbf{h_{22}} & \dots & \mathbf{h_{2h}} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \dots & h_{nh} \end{pmatrix} \times \begin{pmatrix} w^{[2]}_{11} & \mathbf{w^{[2]}_{12}} & \dots & w^{[2]}_{1c} \\ w^{[2]}_{21} & \mathbf{w^{[2]}_{22}} & \dots & w^{[2]}_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ w^{[2]}_{h1} & \mathbf{w^{[2]}_{h2}} & \dots & w^{[2]}_{hc} \end{pmatrix} = \begin{pmatrix} z_{11} & z_{12} & \dots & z_{1c} \\ z_{21} & \mathbf{z_{22}} & \dots & z_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \dots & z_{nc} \end{pmatrix}$$

Example of computing z_{22}

$$z_{22} = h_{21}w^{[2]}_{12} + h_{22}w^{[2]}_{22} + \dots + h_{2h}w^{[2]}_{h2}$$

General example of computing z_{ij}

$$z_{ij} = \sum_{k=1}^{h} h_{ik} \cdot w^{[2]}_{kj}$$

For final layer in regression, where $\mathbb{W}^{[2]}$ is a vector to produce an output column vector $\hat{Y} \in \mathbb{R}^n$ - consisting of scalars for each observation

$$H_{(n,h)} \times W^{[2]}_{(h,1)} = \hat{Y}_{n,1}$$

Each observation x_i has a single $\hat{y}_i$ scalar value. Collectively these form a column vector.

$$H_{(n,h)} \times W^{[2]}_{(h,1)} = \hat{Y}_{(n,1)}$$

$$\begin{pmatrix} h_{11} & h_{12} & \dots & h_{1h} \\ \mathbf{h_{21}} & \mathbf{h_{22}} & \dots & \mathbf{h_{2h}} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1} & h_{n2} & \dots & h_{nh} \end{pmatrix} \times \begin{pmatrix} \mathbf{w^{[2]}_{11}} \\ \mathbf{w^{[2]}_{21}} \\ \vdots \\ \mathbf{w^{[2]}_{h1}} \end{pmatrix} = \begin{pmatrix} y_{11} \\ \mathbf{y_{21}} \\ \vdots \\ y_{n1} \end{pmatrix}$$

Example of computing y_{21}

$$y_{21} = h_{21}w^{[2]}_{11} + h_{22}w^{[2]}_{21} + \dots + h_{2h}w^{[2]}_{h1}$$

General example of computing y_{i1}

$$y_{i1} = \sum_{k=1}^{h} h_{ik} \cdot w^{[2]}_{k1}$$

Feed-forward NN

We have **input data** X with d features ($X = [x_1, x_2, \cdots, x_d]^T$).

For hidden unit h_i:

The NN ties many neurons (hidden units) together that *each are a different transformation of the original features.*

$$\begin{aligned} h_i{}^{[1]}(X) &= g(a_i^{[1]}(X)) \\ &= g\left(b_i^{[1]} + \sum_j^d w_{ij}^{[1]} x_j\right) \end{aligned}$$

- i index: hidden unit.
- j index: original features.

Each hidden unit h_i computes a weighted sum of the all input features i plus a bias, and then applies an activation function g.

- $b_i^{[1]}$ - bias for the ith hidden unit in the first layer (NB: there's a single bias term per output neuron).
- $w_{ij}^{[1]}$ - weight connecting jth input feature to ith hidden unit (number of input dimensions x number of hidden units) - indexed by j and i respectively.

  Weight matrix $W^{[1]}$ contains $d \times H$ weights

  $$\mathbf{W}^{[1]} = \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & \cdots & w_{1H}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & \cdots & w_{2H}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1}^{[1]} & w_{d2}^{[1]} & \cdots & w_{dH}^{[1]} \end{bmatrix}$$

  Where

  - d - number of input features (dimensions)
  - H - number of hidden units in the first layer
  - The weights are indexed by j and i, where:
    - $j = 1, 2, \ldots, d$ indexes the input features.
    - $i = 1, 2, \ldots, H$ indexes the hidden units.

  Connection Directions:

  - Each input feature j is connected to **every** hidden unit i, meaning that $w_{ij}^{[1]}$ exists for all combinations of j and i.
  - Conversely, each hidden unit i receives input from **all** input features j.

Explanation:

Rows: Each row corresponds to the weights associated with one output neuron. For example, for i $(w_{i1}, w_{i2} \ldots, w_{id})$ represents the weights connecting all d inputs to the ith output.

Columns: Each column corresponds to the weights associated with one input feature. For example, column j $((w_{1j}, w_{2j} \ldots, w_{H,j})$ represents the weights connecting the jth input to all h output neurons.

...the hidden activations are crucial for transforming the input data into a representation that can be effectively used by the output layer.

Below, $h^{[1]}$ represents the hidden activations for a single input observation after the transformation through the first layer's weight matrix.

The transformation involves a matrix multiplication of the weight matrix with the input vector, addition of the bias vector, and application of the activation function.

$$\mathbf{h}^{[1]} = \begin{bmatrix} h_1^{[1]} \\ h_2^{[1]} \\ \vdots \\ h_H^{[1]} \end{bmatrix}$$

Each element $h_i^{[1]}$ is the output of the ith neuron in the hidden layer, computed as:

$$h_i^{[1]} = g\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right)$$

- **Activation Function** $g(\cdot)$**:** Introduces non-linearity, enabling the network to model complex relationships.
- **Bias Term** $b_i^{[1]}$**:** Allows each hidden neuron to adjust the activation function independently.
- **Weights** $w_{ij}^{[1]}$**:** Determine the influence of each input feature x_j on the hidden neuron $h_i^{[1]}$.
- **Input Features** x_j**:** The original input data with d features.

Vectorized Representation (single input vector):

Convention: Input vs. Weight Matrix Order

There are two common conventions for writing the linear transformation in a neural network:

1. **Deep Learning / ML library convention (PyTorch, TensorFlow, etc.)**: Inputs are stored as *row vectors*, so we put the input matrix/vector first:

$$X \in \mathbb{R}^{n \times d}, \quad W \in \mathbb{R}^{d \times h}, \quad H = XW \in \mathbb{R}^{n \times h}.$$

   - n = number of samples (rows)
   - d = number of input features
   - h = number of hidden units

   Each row of H is the hidden representation of one input sample.

2. **Math / linear algebra convention**: Inputs are stored as *column vectors*, so we put the weight matrix first:

$$x \in \mathbb{R}^{d \times 1}, \quad W \in \mathbb{R}^{h \times d}, \quad h = Wx \in \mathbb{R}^{h \times 1}.$$

Key Takeaway: Both are mathematically consistent.

- If inputs are row vectors (as in ML libraries), always write XW.
- If inputs are column vectors (as in math derivations), write Wx.

For computational efficiency, the hidden activations for a single observation can be represented in matrix form:

$$\mathbf{h}^{[1]} = g\left((\mathbf{W}^{[1]})^T \mathbf{x} + \mathbf{b}^{[1]}\right)$$

Where:

- $\mathbf{W}^{[1]} \in \mathbb{R}^{d \times H}$ is the weight matrix connecting the input layer to the hidden layer.
- $\mathbf{x} \in \mathbb{R}^{d \times 1}$ is the input feature vector.
- $\mathbf{b}^{[1]} \in \mathbb{R}^{H \times 1}$ is the bias vector for the hidden layer.
- $g(\cdot)$ is applied element-wise to the resulting vector.

Example:
Consider a neural network with:

- $d = 3$ input features
- $H = 4$ hidden units

The hidden activation vector $\mathbf{h}^{[1]}$ would be:

$$\mathbf{h}^{[1]} = \begin{bmatrix} h_1^{[1]} \\ h_2^{[1]} \\ h_3^{[1]} \\ h_4^{[1]} \end{bmatrix}$$

Each $h_i^{[1]}$ is computed as:

$$h_i^{[1]} = g\left(b_i^{[1]} + w_{1i}^{[1]}x_1 + w_{2i}^{[1]}x_2 + w_{3i}^{[1]}x_3\right), \quad \text{for } i = 1, 2, 3, 4$$

... gives us equation of **output layer**: computed by applying an activation function to a weighted sum of the hidden units plus a bias.

$$f(X) = o\left(b^{[2]} + \sum_{i}^{H} w_i^{[2]} h_i^{[1]}\right)$$

- **Bias term** ($b^{[2]}$): just one bias term (a scalar) as we have a single output neuron, so no subscript i for b.
- **Weights term** ($\mathbf{w}_i^{[2]}$):
  - a vector of size $H \times 1$
  - $\mathbf{w}^{[2]} = [w_1^{[2]}, w_2^{[2]}, \cdots, w_H^{[2]}]^T$

$$\mathbf{w}^{[2]} = \begin{bmatrix} w_1^{[2]} \\ w_2^{[2]} \\ \vdots \\ w_H^{[2]} \end{bmatrix}$$

  - connects each hidden unit to the output neuron, determining the influence of each hidden activation on the final output
  - collapses
- **Hidden activations** ($h^{[1]}$)
  - a vector of size $H \times 1$

$$\mathbf{h}^{[1]} = \begin{bmatrix} h_1^{[1]} \\ h_2^{[1]} \\ \vdots \\ h_H^{[1]} \end{bmatrix}$$

  - *from the previous layer* (replace x's role in the initial *input* layer)
    - **input layer** receives raw input features x_j
    - **hidden layer** processes inputs and production activation functions $h_i^{[l-1]}$
    - **output layer**
- **Multiplication process: dot product** ($\mathbf{w}^{[2]T} \times \mathbf{h}^{[1]}$) **-> scalar**)
  - $\mathbf{w}^{[2]} = H \times 1$
  - $\mathbf{w}^{[2]T} = 1 \times H$

$$\mathbf{W}^{[2]T} = \begin{bmatrix} w_{11}^{[2]} & w_{12}^{[2]} & \cdots & w_{1H}^{[2]} \end{bmatrix}$$

  - $\mathbf{h}^{[1]} = H \times 1$

$$\mathbf{h}^{[1]} = \begin{bmatrix} h_1^{[1]} \\ h_2^{[1]} \\ \vdots \\ h_H^{[1]} \end{bmatrix}$$

$$\mathbf{w}^{[2]T} \times \mathbf{h}^{[1]} = \sum_{i}^{H} w_i^{[2]} h_i^{[1]}$$

...this **dot product** produces new **single activation scalar**! This represents the combined activation from the hidden units fed into the output neuron.

- and we are summing over H (number of hidden units in previous layer) rather than d (number of dimensions in the input activation data)
  - The multiplication transforms the dimensions from $H \times 1$ (hidden activations) to 1×1 (output scalar).
  - Whereas the previous multiplication in the hidden layer (the input layer) involves taking an input matrix of dimensions $n \times d$, multiplying by a matrix of dim $d \times H$ resulting in a column vector of hidden activations $h^{[1]} = H \times 1$

... we can write the **complete equation** for the whole network of a single layered network!

$$\begin{aligned} f(X) &= o(a^{[2]}) \\ &= o\left(b^{[2]} + \sum_{i}^{H} w_i^{[2]} h_i(X)\right) \\ &= o\left(b^{[2]} + \sum_{i}^{H} w_i^{[2]} g\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right)\right) \end{aligned}$$

Here, The H hidden units feed into the output layer, which again consists of a linear combination of the hidden units, with an activation function applied.

2.3 NNs basic components

In theory, a single hidden layer with a large number of units has the ability to approximate most functions.

2.3.1 Parameters & hyperparameteters

We learn the weights (b, W) from the data; all the rest are set as hyperparameters.

2.3.2 Back propagation

To learn the weights, we minimize a loss function using gradient descent. When we apply this to neural networks and compute the gradient of the loss function, we call this backpropagation (how we turn info from our loss function into updates of bs and Ws).

2.3.3 Non linear activation functions:

- To prevent collapse into a linear model.
- Can capture complex non-linearities and interaction effects - allows NNs to display any complex function. Whereas in linear regression we have to add the interactions by hand, the NN is able to express them itself.
- Analogous to the brain:
  - values of the activations $h_i(X) = g(a_i(X))$ in each neuron close to 1 are 'firing',
  - values of the activations $h_i(X) = g(a_i(X))$ in each neuron close to 0 are silent.
- For this, we need a function that is *low below a threshold*, and *then high*...
  - Sigmoid
  - ReLU
  - Heavyside function
  - Tanh

Sigmoid ("Logistic activation function" given similarity to logistic function)

$$\begin{aligned} g(z) &= \sigma(z) \\ &= \frac{e^z}{1+e^z} \\ &= \frac{1}{1+e^{-z}} \end{aligned}$$

ReLU (Rectified Linear Unit)

$$g(z) = \max(0, z)$$

Outputs the input directly if it is positive; otherwise, it outputs zero.

$$g(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

Can be computed and stored *more efficiently* than a sigmoid function.

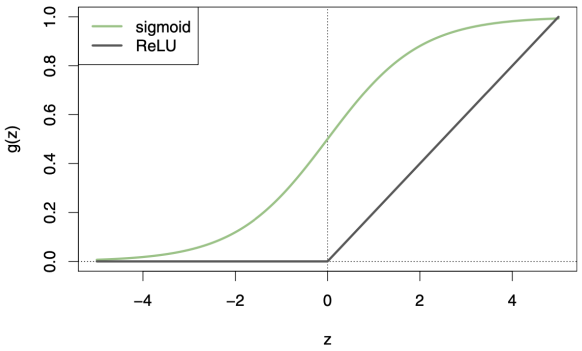


Figure 2.3: Comparing sigmoid vs ReLU

Other activation functions (less common)

Hyperbolic tangent (tanh):

$$g(z) = tanhz = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

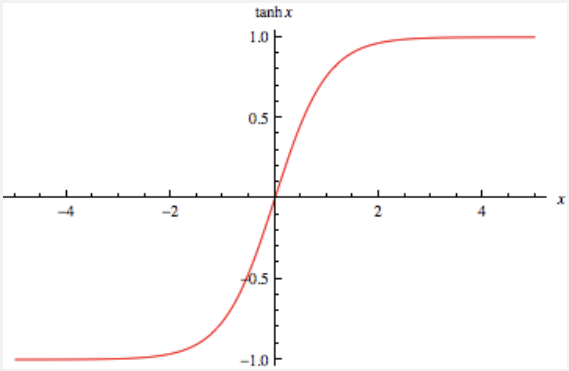


Figure 2.4: Tanh

Heavyside step function:

$$\begin{aligned} g(z) &= H(z) \\ &= I(z > 0) \\ &= \begin{cases} 1 & \text{for } z > 0 \\ 0 & \text{for } z \leq 0 \end{cases} \end{aligned}$$

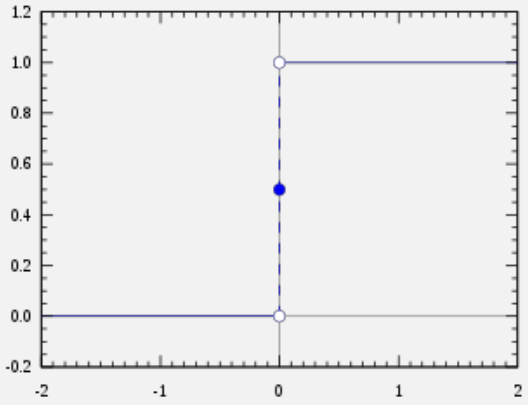


Figure 2.5: Heavyside

Linear (only for output layers)

$$g(z) = z$$

It's just the identity

2.3.4 Output layer activation function

Single output

- **Regression** - we just use the identity $o(a[l]) = a^{[l]}$
- **Binary classification** - we use the sigmoid $o(a^{[l]} = \sigma(a^{[l]})$

Multiclass classification

Produces a vector of probabilities mapping each observation to k possible classes. Requires k nodes in final output layer.

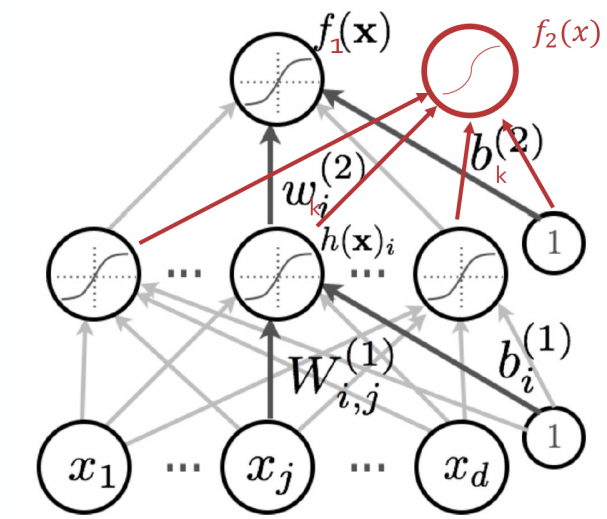


Figure 2.6: for k output nodes; $W^{[2]}$ now has $k \times i$ dim; biases = vector of k

Network with K classes for classification:

$$f_k(X) = o(a^{[L]})_k$$

Where

1. $o(\cdot)$ - activation function for output layer
2. $a_k^{[L]}$ - pre-activation value for the k-th class at final layer L.

1) Pre-activation value

$$a_k^{[l]} = b_k^{[L]} + \sum_i^H w_{ki}^{[L]} h_i^{L-1}(X)$$

Where

- $b_k^{[L]}$ - bias term for the k-th output neuron
- $w_{ki}^{[2]}$ - weights connecting ith hidden unit to kth output unit
- h_i^{L-1} - activation from the ith hidden unit in penultimate layer

2) Output activation

Softmax: $o : \mathbb{R}^K \to (0,1)^K$

(Unlike if we were to use sigmoid activation function again for our output activation, the softmax scales the output so that the vector values sum to 1 ($\sum_{l=1}^{K} o(a^{[2]})_k = 1$); fulfils axioms of probability)

$$o(a_k^{[L]}) = \frac{e^{a_k^{[L]}}}{\sum_{l=1}^{K} e^{a_l^{[L]}}}$$

Where

- K is the total number of classes

Example with two classes: dog vs cat.

If the raw output values are:

$$a^{[2]} = \begin{bmatrix} 1.2 \\ 0.3 \end{bmatrix}$$

The softmax function would compute the probabilities as follows:

$$o(a^{[2]}) = \begin{bmatrix} \frac{e^{1.2}}{e^{1.2}+e^{0.3}} \\ \frac{e^{0.3}}{e^{1.2}+e^{0.3}} \end{bmatrix}$$

Calculating these would yield:

$$o_{\text{dog}} = \frac{e^{1.2}}{e^{1.2}+e^{0.3}} \approx 0.71, \quad o_{\text{cat}} = \frac{e^{0.3}}{e^{1.2}+e^{0.3}} \approx 0.29$$

This means the model predicts a 71% probability that the input is a dog and a 29% probability that it is a cat.

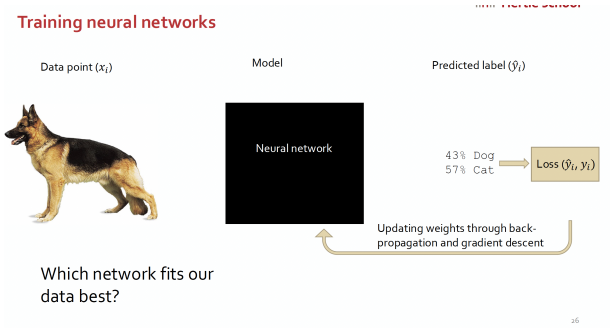


Figure 2.7: The process thus far

2.3.5 Loss function

Ultimately depends on tasks we are looking at / what we want to optimise for.

Loss function for regression

***Sum* of Squared Errors (SSE)**:

$$L(\theta) = L(f(X), y) = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

Or ***Mean* Squared Error (MSE)**:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Other (less common) loss functions:

Mean absolute error
MAE is more robust to outliers because it doesn't square the errors, so large errors have a lesser impact on the overall loss compared to Mean Squared Error (MSE).

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

Mean absolute percentage error
Measures the average absolute percentage difference between the predicted values and the actual values.

$$\text{MAPE} = \frac{100\%}{N}\sum_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

MAPE is useful when you need a relative error measure and is sensitive to the scale of the data.

Mean squared logarithmic error
Measures the squared difference between the logarithms of the predicted and actual value.

$$\text{MSLE} = \frac{1}{N}\sum_{i=1}^{N}(\log(1+y_i) - \log(1+\hat{y}_i))^2$$

MSLE is less sensitive to large differences for high target values and penalizes underestimation more than overestimation.

Cosine similarity
Measures the cosine of the angle between two non-zero vectors of an inner product space.

$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\|\|B\|}$$

Used primarily in text analysis and clustering to measure how similar two vectors are. It ranges from -1 to 1, where 1 means the vectors are perfectly aligned, and 0 means they are orthogonal (no similarity).

Loss function for K-class classification

1. **Sum of Squared Errors** (SSE):

$$L(\theta) = L(f(X), y) = \sum_{i=1}^{N}\sum_{k=1}^{K}(y_{ik} - f_k(x_i))^2$$

We could use this in theory, but it's not ideal.

The *outer sum* iterates over each data sample in the dataset, where N is the total number of samples. Each sample is indexed by i.

The *inner sum* iterates over each class for a given sample i, where K is the total number of classes. Each class is indexed by k.

The actual loss function term computes the loss contribution for class k for sample i:

- y_{ik} = the actual value for class k for sample i. Typically, in a one-hot encoding format, this value is 1 if the sample belongs to class k, and 0 otherwise.
- $f_k(x_i)$ = the predicted probability for class k for sample i, generated by the model.

= computes the squared error for each class for each sample, then sums up these errors across all classes for a given sample and across all samples in the dataset.

2. **Cross-entropy loss**: (more commonly used)

Measures the difference between two probability distributions: the actual distribution (one-hot encoded class labels) and the predicted distribution.

$$L(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} \cdot \log f_k(x_i))$$

It penalizes incorrect class probabilities in a smooth and probabilistic way.

The cross-entropy loss penalizes incorrect predictions based on how confident the model is about its classification.

- y_{ik} - **binary ground truth indicator**
  - Ground truths one-hot encoded → measures difference between estimated distribution from ground truth distribution
  - NB y_{ik} is a scalar: 1 if x_i belongs to class k, and 0 otherwise
- $f_k(x_i)$ - **predicted probability** of class k for sample i
  - (the prob that the model assigns to class k for example x_i.)
- So the **resulting dot product**: $y_{ik} \cdot \log f_k(x_i)$
  → for each sample:
  - only the log probability of the **true class** is considered in the loss.
  - all the other terms are 0'd out.

  (a) If the model assigns a **low probability to the correct** class (i.e., $f_k(x_i) \approx 0$) → logarithm term becomes a large negative value, resulting in a high loss.

  (b) For a **correct and confident prediction** (i.e., $f_k(x_i) \approx 1$), → the logarithm approaches 0, resulting in a low loss.

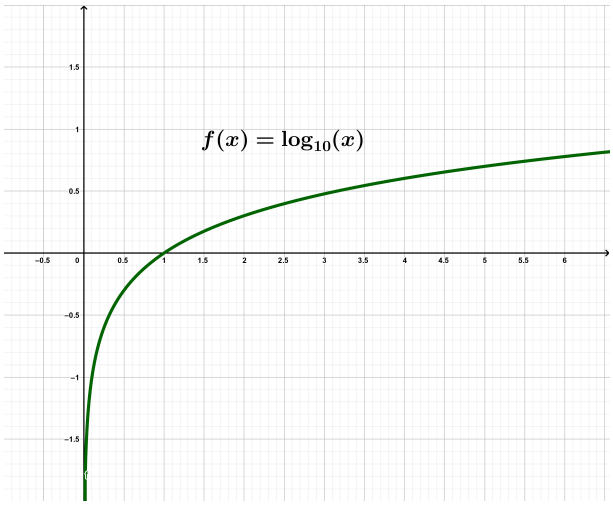


Figure 2.8: x approaches 0 -> y negative; x approaches 1 -> y 0

2.4 Capacity of a NN ('expressiveness')

Expressiveness of a single-layer NN using a single neuron with a sigmoid activation function:

Activation Function:

$$g(a(X)) = \frac{1}{1 + e^{-a(X)}}$$

As before, this gives our single neuron output as:

$$h(X) = g\left(b + \sum_{j=1}^{d} w_j x_j\right)$$

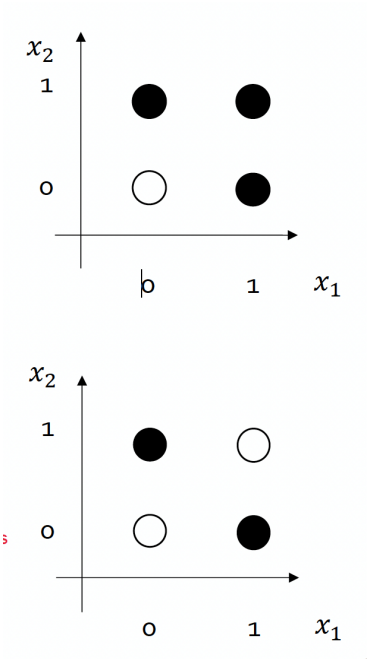


Figure 2.9: Top: linearly separable; bottom, not linearly-separable

Top figure is linearly separable, meaning a single linear decision boundary (e.g., a straight line) can separate the classes; *a single neuron can solve this problem*, as it can create a decision boundary with the weighted sum output formula.

Bottom pattern is not linearly separable (it resembles the XOR problem). *A single neuron cannot learn this problem* (because a single neuron can only create linear decision boundaries. To solve it, we'd need more complex networks with multiple neurons or hidden layers to capture non-linear patterns.

Single neurons are limited to linear decision boundaries. Thus, their expressive capacity is constrained to problems that are linearly separable.

For **non-linearly separable** problems, we need **a combination of neurons (or multi-layer networks)** to build non-linear boundaries, thereby increasing the model's capacity and expressiveness.

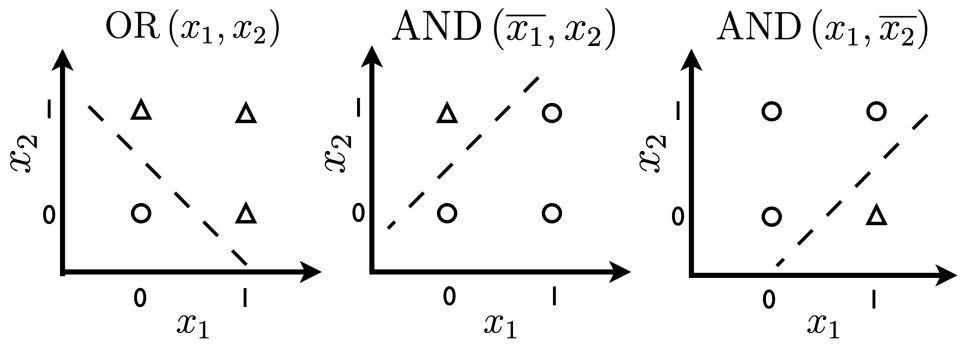


Figure 2.10: Linearly separable problems

A single neuron with sigmoid activation can be interpreted as estimating $P(y = 1|X)$ (logistic classifier).

To solve non-linearly separable problems, we need to **transform input features** to make them linearly-separable.

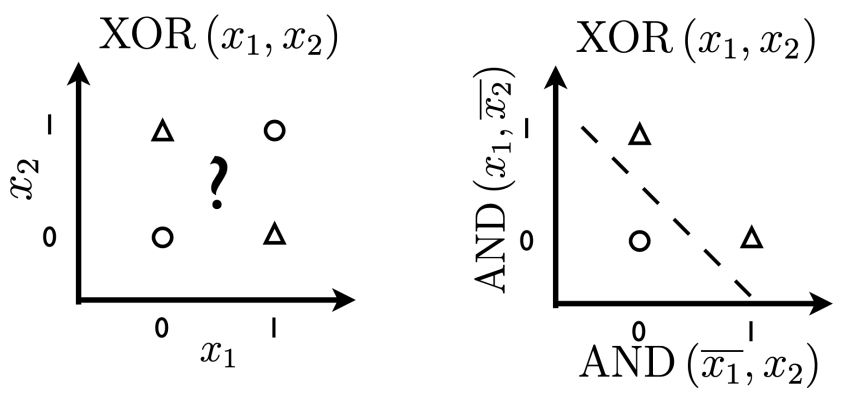


Figure 2.11: Transformation of input features

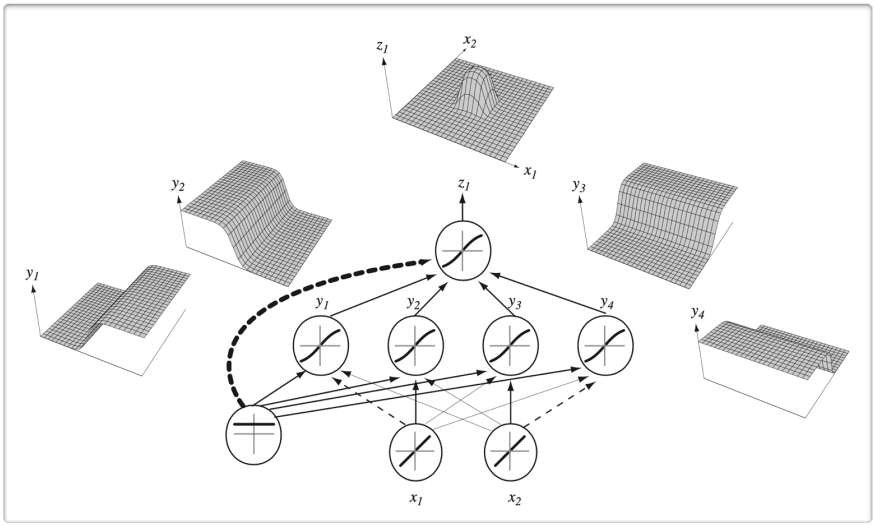


Figure 2.12: Illustration of a how a single-layer NN can represent a non-linearfunction with activations.

This network is trying to learn a function that has a class 1 in the middle but class 0 everywhere else.

- Each of the four neurons learns a separate linearly separable part of this function.
- When these outputs are combined, the network can form a surface that captures the desired complex pattern.

Set up:

- We have a NN with four neurons in a single hidden layer.
- Each neuron takes the input features (x_1, x_2) and applies a linear transformation followed by a non-linear activation function.
- The individual neurons are learning simple linearly separable functions, such as planes or ridges.
- When the outputs of multiple neurons are combined (using the network's final layer), the resulting function can become highly expressive.
- The sum (linear combination?) of the activation functions (with bias terms) allows the network to create complex decision boundaries and represent non-linear functions.
- By adding up these simple patterns, the network can approximate a complex function that is non-linear and multidimensional.

Key Takeaway: Single-layer networks can represent non-linear functions by combining multiple linear neurons. This highlights the importance of activation functions and the combination of neurons for the expressive power of neural networks, even with a single layer.

Universal Approximation Theorem

"A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units"

- Hornik (1991)

However, this does not mean that there is learning algorithm that can find the necessary parameter values.

2.5 Gradient Descent

- In statistical learning we want to find the function (model) $f(X)$ that minimizes the prediction loss (expressed by using one of various loss functions $L(f(X,\theta),Y)$)
- Statistical models are a function of the data and many parameters $f(X,\theta)$.
- In DL, NNs easily have billions of params (weights & biases).
- **Problem:** Following the calculus approach of setting partial derivatives to zero, we would end up with as many equations $\frac{\delta l}{\delta \theta_i} = 0$ as there are parameters (computationally intractable).
- **Solution:** Gradient descent is computationally tractable - approaching the minimum step by step along the direction of steepest descent.
  - We only need to compute the gradient wrt to all our parameters...
  - ... we do not need to then set to zero and solve (a large system of equations) for the parameters.

Instead of

$$\frac{\delta l}{\delta \theta_i} = 0$$

we do

$$\theta \leftarrow \theta - \eta \frac{\delta l}{\delta \theta_i}$$

Example: $f(x,y) = \frac{1}{2}(x^2 + y^2)$.

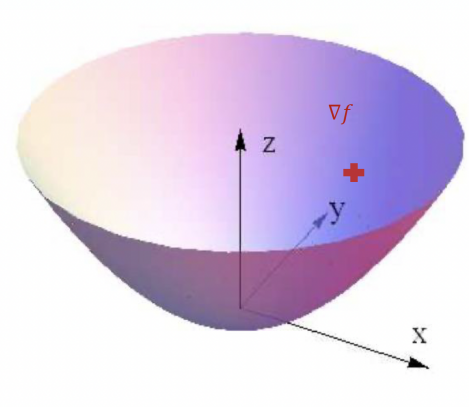


Figure 2.13: The gradient is just a **vector of the partial derivatives**. We need to take the negative of this in order to approach the min.

Iteratively finds the optimal parameters by following the gradient of the loss function, making it a scalable.

NB: only with convex functions would we be guaranteed to 1) move directly on a path to the minimum, 2) reach the global minimum using gradient descent. For non-convex loss functions, gradient descent can be inefficient, or get stuck in local maxima.

Gradient Descent Method:

Given an initial starting point $x^{(0)}$, we want to find a (local) minumum of $f(x)$. We do this until a stopping criterion is fulfilled for each step k:

1. Find the gradient (or search direction) $\Delta x^{(k)} = -\Delta f(x^{(k)})$
2. Choose a stepsize $\eta^{(k)}$ (learning rate)
3. Update with $x^{(k+1)} = x^{(k)} + \eta \Delta x^{(k)}$

NB: x is vector of variables, $\eta > 0$ is a scalar.

Stopping criterion:

- In grad descent, we have that $f(x^{(k+1)}) < f(x^{(k)})$ for some η, except when $x^{(k)}$ is optimal.
- $\rightarrow$ a possible stopping criterion: $\|\nabla f(x^{(k)})\|_2 \leq \epsilon$.

Early Stopping:

- In DL we tend to use *early stopping* instead of a stopping criterion!
- i.e. where we use validation data to determine when to stop.
- So we stop *before* we reach the minimum training error.
- This avoids overfitting.

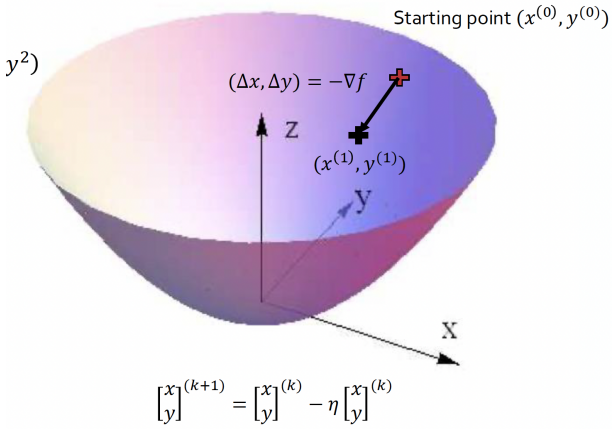


Figure 2.14: *NB, even for fixed η, the steps becomes smaller; as we get closer to 0 the functional values that we plug in will becomes smaller, the gradient evaluated at each successive point becomes smaller. The steps become smaller as the gradients' curve approaches 0, (algebraically the gradient component of the formula gets smaller)*

2.6 Calculating Loss & Backpropagation

2.6.1 Backpropagation process:

Using SGD - optimises loss function by iteratively updating the parameters based on a subset (mini-batch) of the training data.

1. **Forward Pass** - apply model to training data X to get a prediction $\hat{y}$. *See how well the network is currently predicting by calculating the current loss.*
2. **Compute Loss** - apply loss function to get a new loss. Calculates difference between predicted values $\hat{y}$ and actual values y using cross-entropy loss or MSE (as measures of distance). *Tells us how far off the predictions are.*
3. **Backward Pass (Backpropagation) - computes gradient of the loss function wrt each weight.** Determines *how much each weight contributed* to the overall error and calculate its gradient.
   (a) **Calculate the partial derivatives** of the loss function L with respect to each model parameter θ.
      - This involves using chain rule differentiation to propagate the error backwards through the network layers (from output layer to input layer).
      - $\rightarrow$ this gives us the gradient vector ($\frac{\delta L}{\delta \theta}$).
      - The functional form of these gradients is needed to update the weights in the direction that reduces the loss (this is what chain rule derivatives gives us).

   (b) **Plug in *current* parameter values** into $\frac{\delta L}{\delta \theta}$
      - This computes the gradient for current values of the weights and data points/mini-batches.
4. **Update the Parameters** - using update rule from above
   - apply the update rule:

$$\theta \leftarrow \theta - \eta \cdot \frac{\delta L}{\delta \theta}$$

Repeated over many **epochs** (full passes through the dataset) until the model converges to a set of parameters that minimize the loss.

> The hard part is computing the gradient (the backpropagation in step 3).

2.6.2 Computing the Gradient in Backpropagation (step #3)

The function $f(X)$ for a simple single layer neural network with one output (regression) can be represented as:

$$\begin{aligned} f(X) &= o\left(b^{[2]} + \sum_{i}^{H} w_i^{[2]} h_i(X)\right) \\ &= o\left(b^{[2]} + \sum_{i}^{H} w_i^{[2]} g\left(b^{[1]} + \sum_{j}^{d} w_{ij}^{[1]} x_j\right)\right) \end{aligned}$$

Computing gradient of the loss:

Loss function measures error between predicted vs ground truth.

The gradient vector is composed of the following partial derivatives for that function:

$$\nabla L\left(X, y; b_i^{[1]}, w_{ij}^{[1]}, b^{[2]}, w_i^{[2]}\right) = \begin{bmatrix} \frac{\partial L}{\partial b_i^{[1]}} \\ \frac{\partial L}{\partial w_{ij}^{[1]}} \\ \frac{\partial L}{\partial b^{[2]}} \\ \frac{\partial L}{\partial w_i^{[2]}} \end{bmatrix}$$

This is just the gradient of the loss wrt *each parameter in the original loss function* (i.e. the bias & weights across each different layer).

However, these partial derivatives are **not computed directly**. Instead, we apply the chain rule through multiple layers of activations, which involve several intermediate terms (activation function, and preceding layers).

$$\nabla L\left(X, y; b_i^{[1]}, w_{ij}^{[1]}, b^{[2]}, w_i^{[2]}\right) = \begin{bmatrix} \frac{\partial L}{\partial b_i^{[1]}} \\ \frac{\partial L}{\partial w_{ij}^{[1]}} \\ \frac{\partial L}{\partial b^{[2]}} \\ \frac{\partial L}{\partial w_i^{[2]}} \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial b_i^{[1]}} \\ \frac{\partial L}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial w_{ij}^{[1]}} \\ \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial b^{[2]}} \\ \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial w_i^{[2]}} \end{bmatrix}$$

NB: the above simplification is still missing the derivative of the activations, which are necessary to compute the chain rule (below).

Where

- $a^{[1]}$ is the activation in the first layer, which depends on $z^{[1]}$ (the linear combination of the inputs and weights before applying the activation function).
- $a^{[2]}$ is the output activation, which depends on the final layer's linear combination $z^{[2]}$.

Chain rule for Nested Functions of a NN:

$$L = L(o(h(g(x))))$$

Each layer's output depends on the previous layer's activations, and so the gradients must be propagated backwards using the chain rule.

- *(Each weight and bias in a neural network indirectly affects the final output through a series of nested transformations (non-linear activations).)*
- *Thus, to compute the true gradient with respect to a given weight or bias, we need to account for all intermediate activations and their gradients.*

E.g. to calculate $\frac{\delta L}{\delta w_{ij}^{[1]}}$, we need

$$\frac{\partial L}{w_{ij}^{[1]}} = \frac{\partial L}{\partial o} \cdot \frac{\partial o}{\partial h_i} \cdot \frac{\partial h_i}{\partial g} \cdot \frac{\partial g}{\partial w_{ij}^{[1]}}$$

For this, we will need the derivative of (i) the loss function, (ii) the activation functions, and (iii) the output activation.

For example, if the output layer uses softmax for multi-class classification, the partial derivatives of the loss $\frac{\delta L}{\delta w_i^{[2]}}$ will involve the softmax gradient because the loss depends on the probability scores produced by softmax.

Multivariate chain rule

Chains of multivariate functions are common in NNs

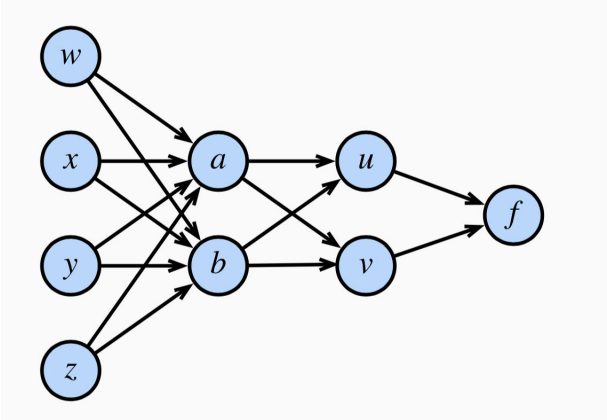


Figure 2.15: $f(u, v)$, but also $f(a, b)$ and $f(w, x, y, z)$

Multivariate chain rule:

$$\frac{\delta}{\delta a} f(u(a,b), v(a,b)) = \frac{\delta f}{\delta u}\frac{\delta u}{\delta a} + \frac{\delta f}{\delta v}\frac{\delta v}{\delta a}$$

Visual intuition, we are just adding all the paths between a and f!

In multivariate setting, we have to consider all the variables: e.g. between f and w:

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial b}\frac{\partial b}{\partial w} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial b}\frac{\partial b}{\partial w}$$

Single-layer neural network unravelled:

$$f(X) = o(a^{[2]})$$

this is the last layer which contains all the layers output * final activation.

$$= o\left(b^{[2]} + \sum_i w_i^{[2]} h^{[1]}\right)$$

here we also have all the hidden units h

$$= o\left(b^{[2]} + \sum_i w_i^{[2]} g\left(a_i^{[1]}\right)\right)$$

here we know what goes into h, it's the activations from the previous layer

$$= o\left(b^{[2]} + \sum_i w_i^{[2]} g\left(b_i^{[1]} + \sum_j w_{ij}^{[1]} x_j\right)\right)$$

We compute the derivatives of each one of these nested components, rather than thinking about it all at once:

1. weight $b_i^{[i]}$ with linear output $o()$:

$$\frac{\partial L(f(X), y)}{\partial b^{[1]}} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial b_i^{[1]}}$$

   - loss wrt to output of the function (last layer) ($\frac{\delta L}{\delta f}$) (also can be written as $o(a^{[2]})$
   - wrt output activation layer $a^{[2]}$
   - activation function is itself a function of the output of the hidden layer before the activation function is applied ($h^{[1]}$)
   - which it itself a function of the previous activation layer $a^{[1]}$ (the weighted inputs from the previous layer)
   - which is itself a function of $b_i^{[i]}$

2. weight w_{ij}

$$\frac{\partial L(f(X), y)}{\partial b^{[1]}} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_i^{[1]}}$$

Many of the partial derivatives are known. Derivatives of common activation functions:

Sigmoid

$$g(z) = \sigma(z) = \frac{e^z}{1 + e^z}$$
$$g'(z) = \sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Tanh

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - \tanh^2(z)$$

ReLu

$$g(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$$

Softmax (for multi-class classification output)

$$o(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

The derivative of the softmax output with respect to z_j is given by:

$$\frac{\partial o(z_i)}{\partial z_j} = \begin{cases} o(z_i)(1 - o(z_j)) & \text{if } i = j \\ -o(z_i)o(z_j) & \text{if } i \neq j \end{cases}$$

NB: has 2 cases (indicates how a change in one input affects the probabilities of all classes):

1. $i = j$ - indicates the influence of class i on itself.

2. $i \neq j$ - indicates influence of class j on class i.

Alternatively, using the Kronecker delta δ_{ij}, the derivative can be expressed as:

$$\frac{\partial o(z_i)}{\partial z_j} = o(z_i)(\delta_{ij} - o(z_j))$$

Worked out example - *Single-layer neural net with squared error loss, linear output fn, and sigmoid activation fn*

Network definition/output:

$$f(X) = b^{[2]} + \sum_{i}^{H} w_i^{[2]} \sigma \left(b_i^{[1]} + \sum_{j}^{d} w_{ij}^{[1]} x_j \right)$$

Loss for one data point (x, y) is given by:

$$L(f(X), y) = (y - f(X))^2$$

Gradient for weight $w_{ij}^{[1]}$:

$$\frac{\partial L(f(X), y)}{\partial w_{ij}^{[1]}} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

1. **Term 1 (Derivative of Loss)**

$$\frac{\partial L}{\partial f} = \frac{\partial}{\partial f} (y - f(x))^2 = -2 (y - f)$$

2. **Term 2 (Derivative of Linear Output)**

$$\frac{\partial f}{\partial a^{[2]}} = \frac{\partial a^{[2]}}{\partial a^{[2]}} = 1$$

3. **Term 3 (Derivative of Output Activation)**

$$\frac{\partial a^{[2]}}{\partial h_i^{[1]}} = \frac{\partial}{\partial h_i^{[1]}} \left(b^{[2]} + \sum_{l}^{H} w_l^{[2]} h_l^{[1]} \right) = w_i^{[2]}$$

NB!!!

4. **Term 4 (Derivative of Activation Function)**

$$\frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} = \frac{\partial \sigma(a_i^{[1]})}{\partial a_i^{[1]}} = \sigma(a_i^{[1]}) \left(1 - \sigma(a_i^{[1]}) \right)$$

5. **Term 5 (Derivative for Hidden Layer Activation)**

$$\frac{\partial a^{[1]}}{\partial w_{ij}^{[1]}} = \frac{\partial}{\partial w_{ij}^{[1]}} \left(b_i^{[1]} + \sum_{m}^{d} w_{im}^{[1]} x_m \right) = x_j$$

NB!!!??????

Putting it together

$$-2(y - f(x)) w_i^{[2]} \sigma(a^{[1]})(1 - \sigma(a)^{[1]}) x_j$$

Gradient update

$$
\begin{aligned}
w_{ij}^{(r+1)} &= w_{ij}^{(r)} + \eta \Delta w_{ij}^{(r)} \\
&= w_{ij}^{(r)} - \eta \left(\frac{\partial L(f(X), y)}{\partial w_{ij}} \right)^{(r)} \\
&* = w_{ij}^{(r)} - \eta \left(-2(y - f(x)) w_i^{[2]} \sigma(a^{[1]})(1 - \sigma(a)^{[1]}) x_j \right)^{(r)}
\end{aligned}
$$

**for a single-layered NN w/ linear output activation*

To get the weight updates for the $r+1$th iteration: we plug in all the values for the rth iteration of:

- the other weights
- all input features of the data
- ground truths & prediction

2.7 Bigger picture

All of this has been for toy example: single layered NN with a linear output activation. In real life we don't have linear output activation, and we use deep networks:

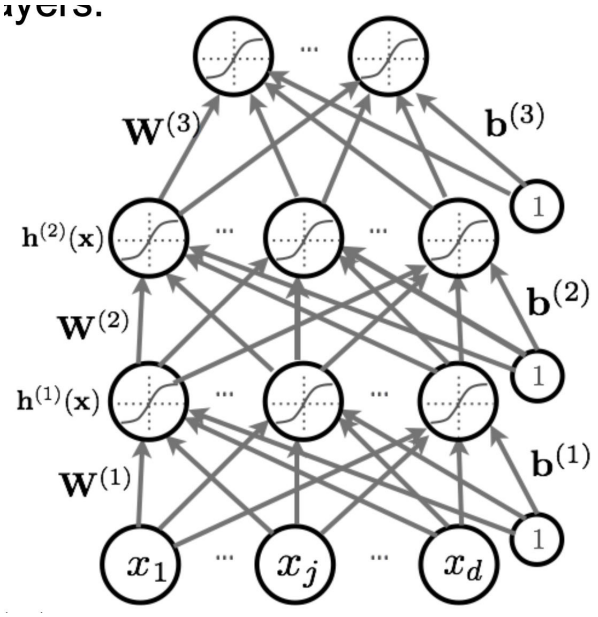


Figure 2.16: Enter Caption

Chapter 3

Deep Neural Networks II

3.1 Overview

- Backpropagation (multi-class classification, several layers)
- Mini batch gradient descent
- Training process
- Vanishing gradient problem

3.2 Backpropagation (continued)

3.2.1 Reminder: single-layered NN

Architecture

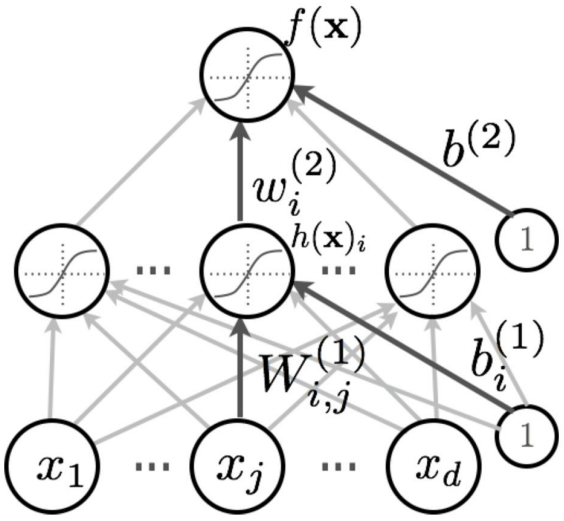


Figure 3.1: single-layered NN

Formal Expression

Reminder: The single-layer, single-output network is expressed as:

$$f(X) = o\left(b^{[2]} + \sum_{i=1}^{H} w_i^{[2]} \sigma\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right)\right)$$

Where:

- $f(X)$ is the output of the network.
- o is the output activation function.
- $b^{[2]}$ and $b_i^{[1]}$ are biases at the second and first layers.
- $w_i^{[2]}$ and $w_{ij}^{[1]}$ are weights at the second and first layers.
- σ is the activation function for the hidden layer.
- x_j are the input features.

Partial derivative:

The partial derivative with respect to the weight $w_{ij}^{[1]}$ is given by:

$$\frac{\partial L(f(X), y)}{\partial w_{ij}^{[1]}} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial a^{[2]}} \cdot \frac{\partial a_i^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

Where:

- L is the loss function.
- $a^{[2]}$ is the pre-activation of the second layer.
- $h_i^{[1]}$ is the output of the activation function for the hidden layer.
- $a_i^{[1]}$ is the pre-activation of the hidden layer.

The weight $w_{ij}^{[1]}$ connects the jth input feature, to the ith neuron.

This weight quantifies the strength and direction of the connection from the jth input feature to the ith node. A higher absolute value of the weight means a stronger influence (pos or neg) of that input feature on the node's output.

The change in the weight $w_{ij}^{[1]}$, which connects the jth input feature to the ith neuron, affects the loss L in the neural network as follows:

$$\frac{\partial L(f(X), y)}{\partial w_{ij}^{[1]}} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial a^{[2]}} \cdot \frac{\partial a_i^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

1. Change in Loss with Respect to Output

$$\frac{\partial L}{\partial f}$$

This measures how the loss L changes with respect to the output f of the neural network. It tells us how sensitive the loss is to changes in the final output.

2. Change in Output with Respect to Preactivation of Second Layer

$$\frac{\partial f}{\partial a^{[2]}}$$

This shows how the output f changes when we adjust the preactivation $a^{[2]}$ of the second layer. Changes in $a^{[2]}$ affect the final output f.

3. Change in Preactivation of Second Layer with Respect to Hidden Layer Output

$$\frac{\partial a_i^{[2]}}{\partial h_i^{[1]}}$$

This represents how the preactivation $a_i^{[2]}$ of the second layer changes when we alter the output $h_i^{[1]}$ from the ith neuron in the first hidden layer.

4. Change in Hidden Layer Output with Respect to Activation of First Layer

$$\frac{\partial h_i^{[1]}}{\partial a_i^{[1]}}$$

This indicates how the output $h_i^{[1]}$ from the ith neuron in the first hidden layer is affected by changes in its preactivation $a_i^{[1]}$.

5. Change in Activation of First Layer with Respect to Weight

$$\frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

Finally, this describes how the preactivation $a_i^{[1]}$ of the ith neuron in the first layer changes when we tweak the weight $w_{ij}^{[1]}$.

Putting It All Together The entire expression shows how a change in the weight $w_{ij}^{[1]}$ influences the loss L through a chain of relationships:

- Starting from the loss, we see how sensitive it is to changes in the output of the network.
- Next, we track how the output depends on the preactivation of the second layer.
- Then, we look at how the preactivation of the second layer depends on the output of the first layer's neuron.
- We continue with how the output of that neuron depends on its preactivation.

NB: the preactivation of the second layer $a^{[2]}$ doesn't have an index in this case is because it's dealing with a single-output network. In a network with only one output neuron, the second layer contains a single preactivation value. Therefore, the preactivation $a^{[2]}$ corresponds to the input to the final activation function before generating the single output.

Gradient update:

We update the weights using gradient descent and learning rate η:

$$w_{ij}^{(r+1)} = w_{ij}^{(r)} + \eta \Delta w_{ij}^{(r)} = w_{ij}^{(r)} - \eta \left(\frac{\partial L(f(X), y)}{\partial w_{ij}} \right)^{(r)}$$

Where:

- $w_{ij}^{(r)}$ is the weight at the r^{th} iteration.
- η is the learning rate.
- $\Delta w_{ij}^{(r)}$ is the change in weight based on the gradient at the r^{th} iteration. (A scalar)

3.2.2 Multivariate chain rule

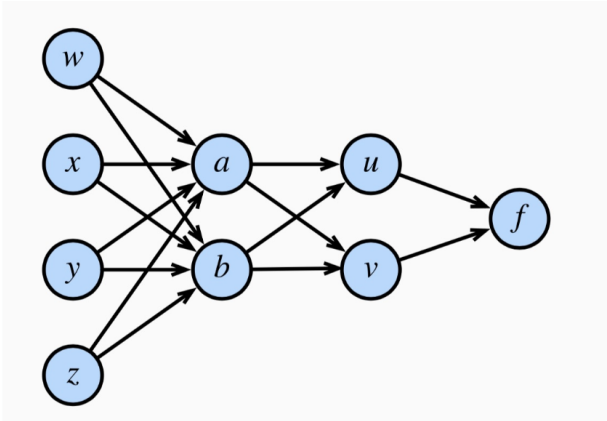


Figure 3.2: Multivariate chain rule, where notes=variables, edges=functional dependence

Chains of multivariate functions are common in neural networks. The chain rule is a key ingredient for backpropagation. Consider a function f that depends on $u(a, b)$ and $v(a, b)$, where a and b themselves depend on other variables.

Multivariate chain rule:

$$\frac{\partial}{\partial a} f(u(a, b), v(a, b)) = \frac{\partial f}{\partial u} \frac{\partial u}{\partial a} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial a}$$

Interpretation:
We want to calculate how the function $f(u(a,b), v(a,b))$, which depends on u and v, changes when we change a.

- The first term, $\frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial a}$, represents the change in f caused by changes in u, which are themselves caused by changes in a.
- The second term, $\frac{\partial f}{\partial v} \cdot \frac{\partial v}{\partial a}$, represents the change in f caused by changes in v, which are also caused by changes in a.

Overall Explanation:
The equation tells us that a small change in a can affect f in two ways:

- **Through** u: A change in a causes u to change, which affects f.
- **Through** v: A change in a also causes v to change, which affects f.

Therefore, the total change in f with respect to a is the sum of these two effects: the change in f through u, and the change in f through v.

Example: How many terms do we need to add up to compute $\frac{\partial f}{\partial z}$?
Using the multivariate chain rule, we can break down $\frac{\partial f}{\partial z}$ as follows:

$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a}\frac{\partial a}{\partial z} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}\frac{\partial a}{\partial z} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial b}\frac{\partial b}{\partial z} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial b}\frac{\partial b}{\partial z}$$

This gives the complete derivative of f with respect to z, accounting for all the possible paths through which z influences f via a, b, u, and v.

3.3 Multiple Output Nodes ($k > 1$)

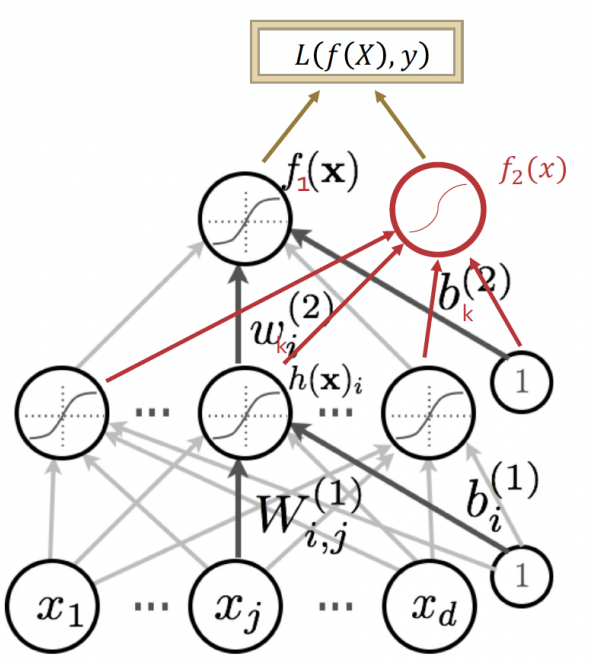


Figure 3.3: 2 output nodes ($k = 2$)

The function for the k-th output node in a single-layer neural network is defined as:

$$f_k(\mathbf{X}) = o\left(b_k^{[2]} + \sum_{i=1}^{H} w_{ki}^{[2]} \sigma\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right)\right)_k$$

Where:

- o is the output activation function (often softmax for classification).
- $b_k^{[2]}$ is the bias for the k-th output node in the second layer.
- $w_{ki}^{[2]}$ is the weight connecting the i-th node in the hidden layer to the k-th output node.
- σ is the activation function of the hidden layer (e.g., ReLU, Sigmoid).
- $b_i^{[1]}$ is the bias of the i-th node in the hidden layer.
- $w_{ij}^{[1]}$ is the weight connecting the j-th input node to the i-th hidden node.
- x_j is the input feature.

3.3.1 Cross-Entropy Loss:

Measures the difference ("distance") between the predicted probability distribution output by the model and the true probability distribution represented by the labels.

The cross-entropy loss for multi-class classification is defined as:

$$L(f(\mathbf{X}), \mathbf{y}) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(x_i)$$

$f_k(x_i)$ is the prob that the model assigns to class k for example x_i.

y_{ik} is 1 if x_i belongs to class k, and 0 otherwise

Where:

- N is the number of training examples.
- K is the number of classes.
- y_{ik} is the true label (one-hot encoded) for example i and class k.
- $f_k(x_i)$ is the predicted probability of class k for example i.

If the model predicts a class with high confidence (e.g., 0.9) and that class is the correct one, the loss is small.

If the model predicts the wrong class with high confidence (e.g., 0.9), the loss is large.

If the model spreads its probabilities evenly across the classes (e.g., 0.25 for each class in a 4-class problem), it will still incur a significant loss unless the true class is one of the higher probabilities.

1. Logarithmic Penalty:

NB ***only the term corresponding to the correct class contributes to the loss***

The logarithmic part in the cross-entropy loss function, $\log f_k(x_i)$:

- If the model predicts a **high probability** for the **correct class**, i.e., $f_k(x_i) \approx 1$, then $\log f_k(x_i)$ will be close to 0. This results in a small loss because the prediction is accurate.

$$\log(1) = 0$$

- However, if the model assigns a **low probability** to the **correct class**, i.e., $f_k(x_i)$ is close to 0, the logarithmic value $\log f_k(x_i)$ becomes a large negative number. This results in a large loss, reflecting the model's poor prediction.

$$\log(0.01) \approx -4.605$$

  The steeper the drop in probability (approaching 0), the higher the negative value of the logarithm, and hence the larger the penalty. The logarithmic function ensures that predictions which are confidently wrong incur a significant penalty, encouraging the model to increase the probability of the correct class.

2. One-Hot Encoding:

True labels y_{ik} are typically represented using one-hot encoding.

- For each example i, the true label is a vector where one entry corresponding to the correct class is set to 1, and all others are 0. In other words, if the true class for example i is class k, then $y_{ik} = 1$ and $y_{ij} = 0$ for all $j \neq k$.

$$\mathbf{y}_i = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{(if the correct class is the third one)}$$

- This encoding ensures that the **loss function only considers the predicted probability for the correct class**.

  Only the term corresponding to the correct class contributes to the loss because $y_{ik} = 1$ for the correct class and 0 for all other classes. The cross-entropy loss is thus focused on maximizing the probability of the correct class while ignoring the others.

Combined Effect:
The combination of the logarithmic penalty and one-hot encoding has the following effects:

- It strongly penalizes predictions that assign low probabilities to the correct class.
- It emphasizes improving the model's confidence in the correct class.
- Predictions that are close to 1 for the correct class yield low loss, encouraging high confidence when the model is correct.

3.3.2 Gradient Calculation:

The gradient of the loss function with respect to the weights is given by:

$$\nabla L(f(\mathbf{X}), \mathbf{y})$$

The output of the neural network is:

$$f(\mathbf{X}) = (f_1(\mathbf{X}), f_2(\mathbf{X}), \ldots, f_K(\mathbf{X}))$$

Multivariate Chain Rule

Multivariate Chain Rule (General)...

$$\frac{\partial}{\partial a} f(u(a,b), v(a,b)) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}$$

...Applying the Chain Rule to Multi-Class Classification

To compute the derivative of the loss function $L(f(\mathbf{X}), \mathbf{y})$ with respect to a weight w_{ij}:

$$\frac{\partial L(f(\mathbf{X}), \mathbf{y})}{\partial w_{ij}} = \sum_{k=1}^{K} \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial a_k^{[2]}} \cdot \frac{\partial a_k^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

Breaking this down:

- $\frac{\partial L}{\partial f_k}$: Gradient of the loss with respect to the output of the k-th output node.
- $\frac{\partial f_k}{\partial a_k^{[2]}}$: Derivative of the output activation function with respect to the input to the final layer.
- $\frac{\partial a_k^{[2]}}{\partial h_i^{[1]}}$: Derivative of the final layer input with respect to the hidden layer activations.
- $\frac{\partial h_i^{[1]}}{\partial a_i^{[1]}}$: Derivative of the hidden layer activation function with respect to the pre-activation value.
- $\frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$: Derivative of the pre-activation value with respect to the weight connecting input j to hidden node i.

Cross-Entropy's Loss Gradient (simplifies!)

Multi-output's Cross Entropy's Loss Gradient has a nice simplification - makes it easy to compute!

> *NB - context: this is the **first link** in the multivariate chain rule!*

Plugging in the cross-entropy loss formula into the multivariate chain rule...

...gives us the gradient of the cross-entropy loss with respect to the output f_k is:

$$\frac{\partial L(f(\mathbf{X}), \mathbf{y})}{\partial f_k} = -\frac{\partial}{\partial f_k} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(x_i)$$

NB this is the loss for every single observation, across every class

This simplifies to:

$$= -\sum_{i=1}^{N} y_{ik} \frac{1}{f_k(x_i)}$$

Here, y_{ik} is 1 if example i belongs to class k, and 0 otherwise.

Combined expression of Softmax and Cross-Entropy

Often, the cross-entropy loss is combined with the softmax activation function, simplifying the gradient computation to:

$$\frac{\partial L}{\partial a_k}$$

NB: This is because the logarithmic term from the cross-entropy and the exponential term from the softmax cancel out...

$$\text{Softmax function: } f_k(\mathbf{x}) = \frac{e^{a_k}}{\sum_{j=1}^{K} e^{a_j}}$$

$$\text{SIMPLIFIED Cross-entropy loss: } L(f(\mathbf{X}), \mathbf{y}) = -\sum_{i=1}^{N} y_{ik} \frac{1}{f_k(x_i)}$$

...leaving a clean and simple expression for the gradient (wrt to the logits a_k).

$$\frac{\partial L}{\partial a_k} = f_k(x_i) - y_{ik}$$

*This is a clean and easy to compute version of the **first link** in the multivariate chain*

This simplifies backpropagation as we can directly compute $\frac{\partial L}{\partial a_k}$ without separately computing derivatives for the softmax function and cross-entropy loss.

$$\frac{\partial L}{\partial a_k} = f_k(x_i) - y_{ik}$$

- $f_k(x_i)$ is the predicted prob that the model assigns to class k for input x_i
- y_{ik} is the true label for class k (1 or 0)

This simplification is one of the reasons why softmax and cross-entropy are often used together for classification tasks, as it makes the backpropagation step computationally efficient and easy to implement.

Proof: how the cross-entropy loss and softmax function are combined and how the derivative with respect to the logits simplifies. This process leads to an elegant expression for the gradient, which is efficient for backpropagation.

The cross-entropy loss for multi-class classification is given by:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log f_k(x_i)$$

$$f_k(x_i) = \frac{e^{a_k}}{\sum_{l=1}^{K} e^{a_l}}$$

Here, a_k are the logits, or raw outputs, of the final layer before applying softmax.
Substituting Softmax into Cross-Entropy Loss

Substituting the softmax expression for $f_k(x_i)$ into the cross-entropy loss:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \log\left(\frac{e^{a_k}}{\sum_{l=1}^{K} e^{a_l}}\right)$$

Expanding this gives:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \left(\log e^{a_k} - \log \sum_{l=1}^{K} e^{a_l}\right)$$

Simplifying the logarithmic terms:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{N} \left(\log \sum_{l=1}^{K} e^{a_l} - \sum_{k=1}^{K} y_{ik} a_k\right)$$

Derivative of Loss with Respect to Logits

To compute the gradient of the loss with respect to the logit a_k, apply the chain rule:

$$\frac{\partial L}{\partial a_k} = \frac{\partial}{\partial a_k} \sum_{i=1}^{N} \left(\log \sum_{l=1}^{K} e^{a_l} - \sum_{k=1}^{K} y_{ik} a_k\right)$$

This derivative splits into two parts: 1. The derivative of the normalization term $\log \sum_{l=1}^{K} e^{a_l}$:

$$\frac{\partial}{\partial a_k} \log \sum_{l=1}^{K} e^{a_l} = \frac{e^{a_k}}{\sum_{l=1}^{K} e^{a_l}} = f_k(x_i)$$

2. The derivative of the second term, involving the true labels:

$$\frac{\partial}{\partial a_k} \left(-\sum_{k=1}^{K} y_{ik} a_k\right) = -y_{ik}$$

Combining the Derivatives

Thus, combining the two derivative terms, the gradient of the loss function with respect to the logit a_k becomes:

$$\frac{\partial L}{\partial a_k} = f_k(x_i) - y_{ik}$$

Summary Intuition:

By combining the softmax function and cross-entropy loss, the gradient simplifies to:

$$\frac{\partial L}{\partial a_k} = f_k(x_i) - y_{ik}$$

- **Correct Class:** When $y_{ik} = 1$ (the true class for input x_i), the gradient is $f_k(x_i) - y_{ik}$
  - This tells us how far off the model's prediction $f_k(x_i)$ is from the true value. If the predicted probability is close to 1, the gradient will be small, resulting in minor adjustments. If the probability is far from 1, the gradient will be larger, leading to a more significant update.
- **Incorrect Classes**: When $y_{ik} = 0$ for incorrect classes, the gradient is $f_k(x_i)$.
  - If the model assigns a high probability to an incorrect class, this will result in a large gradient, causing the model to reduce that probability during training.

This simplification makes backpropagation efficient, as the model adjusts predictions based on how far off they are from the true labels. It avoids explicitly computing the derivative of the softmax function, making training faster and more stable.

3.3.3 Two hidden layer NNs (Multilayer Perceptron)

Backpropagation in 2 hidden layers:

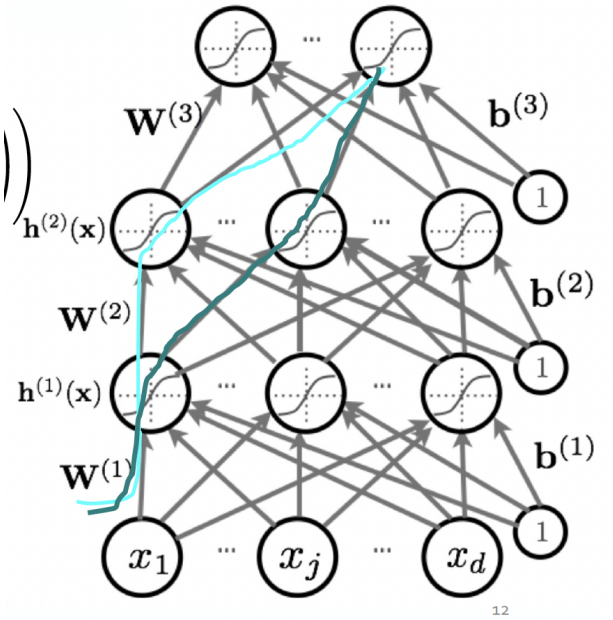


Figure 3.4: Mutlilayer Perceptron (2 hidden layers)

The network's output for class k is computed as:

$$f_k(\mathbf{x}) = o\left(b_k^{[3]} + \sum_{l=1}^{H^{[2]}} w_{kl}^{[3]} g\left(b_l^{[2]} + \sum_{i=1}^{H^{[1]}} w_{li}^{[2]} g\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right)\right)\right)$$

The General Form of the chain rule:

The gradient of the loss function with respect to the weights in the first layer $w_{ij}^{[1]}$ **generically given** as:

$$\frac{\partial L(f(\mathbf{x}), \mathbf{y})}{\partial w_{ij}^{[1]}} = \sum_{k=1}^{K} \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial w_{ij}^{[1]}}$$

This highlights the beginning and end of the backpropagation process.

It gives a compact representation that shows how the loss depends on the weights in the first layer, without yet detailing the intermediate layers.

*NB: this expression is still v high level - it just provides the "start" and "finish" of the chain. This is invariant to how many hidden layers there are in the network (hence **generic**)*

Once we know how many hidden layers there are, we will use the chain rule to expand the second term $\frac{\partial f_k}{\partial w_{ij}^{[1]}}$.

When we expand, $\frac{\partial f_k}{\partial w_{ij}^{[1]}}$, it involves a series of intermediate layers (which are/depend on the number of hidden layers).

Full Expansion with Multivariate chain rule:

Expanding the Gradient of f_k with Respect to $w_{ij}^{[1]}$

To expand the term $\frac{\partial f_k}{\partial w_{ij}^{[1]}}$ using the multivariate chain rule, we have:

$$\frac{\partial f_k}{\partial w_{ij}^{[1]}} = \frac{\partial f_k}{\partial a_k^{[3]}} \cdot \sum_{l=1}^{H^{[2]}} \frac{\partial a_k^{[3]}}{\partial h_l^{[2]}} \cdot \frac{\partial h_l^{[2]}}{\partial a_l^{[2]}} \cdot \sum_{i=1}^{H^{[1]}} \frac{\partial a_l^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

In this expansion:

- The second hidden layer is accounted for by the terms:

$$\frac{\partial a_k^{[3]}}{\partial h_l^{[2]}} \quad \text{and} \quad \frac{\partial h_l^{[2]}}{\partial a_l^{[2]}}$$

- The first hidden layer is accounted for by the term:

$$\frac{\partial h_i^{[1]}}{\partial a_i^{[1]}}$$

$$\frac{\partial L}{\partial w_{ij}^{[1]}} = \sum_{k=1}^{K} \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial a_k^{[3]}} \cdot \sum_{l=1}^{H^{[2]}} \frac{\partial a_k^{[3]}}{\partial h_l^{[2]}} \cdot \frac{\partial h_l^{[2]}}{\partial a_l^{[2]}} \cdot \sum_{i=1}^{H^{[1]}} \frac{\partial a_l^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

Using the full chain rule, this captures the contributions from all intermediate layers.

3.4 Vectorization

Purpose: create efficiency in terms of compute.

3.4.1 Basic concept

Scalar Multiplications

When performing scalar multiplications, the sum of element-wise multiplications between two vectors $\mathbf{w}$ and $\mathbf{x}$ of length d is given by:

$$\sum_{j=1}^{d} w_j \cdot x_j$$

For the computer, this is typically implemented using a loop:

```
sum = 0
for j in range(d):
    result = w[j] * x[j]
    sum = sum + result
```

This approach involves iterating over each element, performing a multiplication, and then summing the results sequentially. This means that the computer repeatedly executes the same instructions for each element.

Vector Multiplications

In contrast, vector multiplication, or the dot product of two vectors, is computed as:

$$\sum_{j=1}^{d} w_j \cdot x_j = \mathbf{w} \cdot \mathbf{x}$$

For the computer, this can be implemented efficiently using vectorized operations. In Python (using NumPy), this is written as:

```
sum = np.dot(w, x)
```

Advantages of Vectorization (for computation)

Vectorization brings several benefits:

- **No repetitions of instructions:** Unlike looping, vectorization doesn't require repeatedly executing the same instructions for each element.
- **Parallel execution:** Vectorized operations allow the computer to execute multiple operations simultaneously, leveraging modern CPU and GPU architectures.

3.4.2 Vectorizing the Neural Network

Single-Layer Neural Network:

For a single-layer neural network, the output for class k, $f_k(\mathbf{X})$, can be written as:

$$f_k(\mathbf{X}) = o\left(b_k^{[2]} + \sum_{i=1}^{H} w_{ki}^{[2]} g\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right)\right)_k$$

Vectorized Form:

This equation can be vectorized to efficiently compute the output for all classes at once:

$$\mathbf{f}(\mathbf{x}) = o\left(\mathbf{b}^{[2]} + \mathbf{W}^{[2]}\mathbf{h}^{[1]}\right)$$

Where:

- $f(x) \in \mathbb{R}^K$
- $\mathbf{b}^{[2]} \in \mathbb{R}^K$ is the bias vector for the second layer,
- $\mathbf{W}^{[2]} \in \mathbb{R}^{K \times H}$ is the matrix of weights for the second layer,
- $\mathbf{h}^{[1]} \in \mathbb{R}^H$ is the vectorized output of the hidden layer ($\mathbf{h}^{[1]} = g\left(\mathbf{b}^{[1]} + \mathbf{W}^{[1]}\mathbf{x}\right)$)

For the hidden layer:

$$\mathbf{h}^{[1]} = g\left(\mathbf{b}^{[1]} + \mathbf{W}^{[1]}\mathbf{x}\right)$$

Where:

- $\mathbf{h}^{[1]} \in \mathbb{R}^H$
- $\mathbf{b}^{[1]} \in \mathbb{R}^H$ is the bias vector for the first layer,
- $\mathbf{W}^{[1]} \in \mathbb{R}^{H \times d}$ is the matrix of weights for the first layer,
- $\mathbf{x} \in \mathbb{R}^d$ is the input vector.

$$\underset{K}{\boldsymbol{f}(x)} = o\big(\underset{K}{\boldsymbol{b}^{[2]}} + \underset{K\times H}{W^{[2]}}\underset{H}{\boldsymbol{h}^{[1]}}\big)$$

$$\underset{H}{\boldsymbol{h}^{[1]}} = g\big(\underset{H}{\boldsymbol{b}^{[1]}} + \underset{H\times d}{W^{[1]}}\underset{d}{\boldsymbol{x}}\big)$$

Figure 3.5: with associated dimensions subscripted

More Compact Representation:

To further simplify, we can incorporate the bias terms into the weight matrices by adding an additional dimension to the input vector:

$$f(\mathbf{x}) = o\left(\mathbf{W}^{[2]}g\left(\mathbf{W}^{[1]}\mathbf{x}\right)\right)$$

Where the bias terms are now included in the weight matrices.

$$\boldsymbol{f(x) = o\left(W^{[2]} \quad g\left(W^{[1]} \quad \boldsymbol{x}\right)\right)}$$
$$\boldsymbol{K \qquad K\times(H+1) \qquad (H+1)\times(d+1) \qquad (d+1)}$$

$$f(\mathbf{x}) = o\left(\mathbf{W}^{[2]} g\left(\mathbf{W}^{[1]}\mathbf{x}\right)\right)$$

$$\begin{aligned} K \qquad & \mathbf{W}^{[2]} \in \mathbb{R}^{K\times(H+1)} \\ K\times(H+1) \qquad & \mathbf{W}^{[1]} \in \mathbb{R}^{(H+1)\times(d+1)} \\ (H+1)\times(d+1) \qquad & \mathbf{x} \in \mathbb{R}^{d+1} \end{aligned}$$

Additionally, the input vector is extended by adding a component 1, so that:

$$\mathbf{x} = (1, x_1, \ldots, x_d) \in \mathbb{R}^{d+1}$$

Matrix Representation of the Weight Matrices:

With this compact representation, the first layer's weight matrix $\mathbf{W}^{[1]}$ now includes the bias terms:

$$\mathbf{W}^{[1]} = \begin{bmatrix} b_1 & w_{11}^{[1]} & w_{12}^{[1]} & \ldots & w_{1d}^{[1]} \\ b_2 & w_{21}^{[1]} & w_{22}^{[1]} & \ldots & w_{2d}^{[1]} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_H & w_{H1}^{[1]} & w_{H2}^{[1]} & \ldots & w_{Hd}^{[1]} \end{bmatrix}$$

This compact representation makes it easier to compute the outputs in matrix form, improving computational efficiency.

Forward Pass: 2-Hidden-Layer Neural Network

For a two-hidden-layer neural network, the forward pass can be written as:

$$f(\mathbf{x}) = o\left(\mathbf{W}^{[3]} g\left(\mathbf{W}^{[2]} g\left(\mathbf{W}^{[1]}\mathbf{x}\right)\right)\right)$$

Forward Pass: L-Hidden-Layer Neural Network

More generally, for an L-layer neural network (MLP), the forward pass is written as:

$$f(\mathbf{x}) = o\left(\mathbf{W}^{[L]} g\left(\mathbf{W}^{[L-1]} \ldots g\left(\mathbf{W}^{[1]}\mathbf{x}\right)\right)\right)$$

This can also be written as:

$$f(\mathbf{x}) = o\left(\mathbf{W}^{[L]} g\left(\mathbf{W}^{[L-1]} \ldots g\left(\mathbf{a}^{[1]}\right)\right)\right)$$

Since:

$$\mathbf{a}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}$$

So, more generally:

$$\begin{aligned} f(\mathbf{x}) &= o\left(\mathbf{W}^{[L]} g\left(\mathbf{W}^{[L-1]} \mathbf{h}^{[L-2]}\right)\right) \\ &= o\left(\mathbf{a}^{[L]}\right) \end{aligned}$$

Note that across literature and in problem sets:

- $\mathbf{a}^{[l]}$ is sometimes referred to as $\mathbf{z}^{[l]}$.
- $\mathbf{h}^{[l]}$ is sometimes referred to as $\mathbf{a}^{[l]}$.

3.4.3 Backpropagation Vectorized

Backpropagation becomes more involved when vectorized, requiring more advanced linear algebra. However, the solution can be written compactly using the error signal.

The Error Signal

denoted as $\delta^{[L]}$ for layer l

is a vector that quantifies how much the output of a neural network differs from the expected outcome (the target value).

It plays a crucial role in the backpropagation algorithm, as it indicates the **direction** and **magnitude** of changes needed for the weights in the network to minimize the loss function.

Purpose: The error signal is used to calculate how much each weight should be adjusted during training to reduce the overall loss of the network. It helps propagate the error from the output layer back through the network to the input layer.

Computation: For the last layer (output layer), the error signal can be computed using the derivative of the loss function with respect to the output of that layer, combined with the derivative of the activation function. For example, in a softmax and cross-entropy scenario, it captures how wrong the predictions are relative to the true labels.

$$\begin{aligned} \delta^{[L]} &\equiv \nabla_{\mathbf{a}^{[L]}} L \\ &\equiv \frac{\partial L}{\partial f} \cdot o'\left(\mathbf{a}^{[L]}\right) \end{aligned}$$

The the gradient $\nabla_{\mathbf{a}^{[L]}} L$ represents how much the loss L changes wrt to the preactivation values $a^{[L]}$ at the final layer. This is also referred to as the error signal $\delta^{[L]}$.

Last Layer

The gradient of the loss L with respect to the activations at the final layer $\mathbf{a}^{[L]}$ is:

$$\nabla_{\mathbf{a}^{[L]}} L = \frac{\partial L}{\partial f} \cdot o'\left(\mathbf{a}^{[L]}\right) \equiv \delta^{[L]}$$

describes the gradient of the loss L wrt the activations at the final layer $a^{[L]}$.

- $\frac{\partial L}{\partial f}$: represents how much the loss changes with respect to the final output of the network $f(x)$.

  - For example, if we are using cross-entropy loss combined with softmax, this would be the difference between the predicted probability and the true label.

- $o'\left(\mathbf{a}^{[L]}\right)$: derivative of the output activation function o, evaluated at the pre-activation values $a^{[L]}$ of the final layer of the final layer.

  - If o is is softmax or sigmoid, this would be the derivative of those functions

The gradient with respect to the weights $\mathbf{W}^{[L]}$ is:

$$\begin{aligned}\nabla_{\mathbf{W}^{[L]}} L &= \frac{\partial L}{\partial f} \cdot o'\left(\mathbf{a}^{[L]}\right) \frac{d\mathbf{a}^{[L]}}{d\mathbf{W}^{[L]}} \\ &= \delta^{[L]} \mathbf{h}^{[L-1]}\end{aligned}$$

describes the gradient of the loss with respect to the weights in the final layer $W^{[L]}$.

The new terms here:

- $\frac{d\mathbf{a}^{[L]}}{d\mathbf{W}^{[L]}}$: this term is the derivative of the pre-activation values $a^{[L]}$ ((the weighted sum before applying the activation function) with respect to the weights $W^{[L]}$.

  - since $a^{[L]} = W^{[L]} h^{[L-1]} + b^{[L]}$, the derivative $\frac{d\mathbf{a}^{[L]}}{dW^{[L]}}$ is simply $h^{[L-1]}$, the activations from the previous layer.

This means the gradient of the activations $a^{[L]}$ wrt to the weights $W^{[L]}$ is simply the activations from the previous layer.

Gradient with respect to the weights in the final layer

$$\nabla_{\mathbf{W}^{[L]}} L = \delta^{[L]} \cdot \mathbf{h}^{[L-1]}$$

This is important because it tells us how the weights in the final layer should be adjusted during backpropagation:

- $d^{[L]}$ is the error signal how much (the prediction differs from the true value)
- $h^{[L-1]}$ is the activation from the previous layer, which is used to update the weights based on the error.

Intuition:

- The final output of the neural network depends on the weights of the last layer. Therefore, if the network makes a mistake (i.e., the loss is high), the gradient $\nabla_W^{[L]} L$ will guide how much the weights in the last layer $W^{[L]}$ should be updated the reduce the error.
- **Error Signal component:** the larger the error signal $\delta^{[L]}$, the more significant the weight update.
- **Previous Layer's activations component**: The magnitude of the update is also scaled by the activations $h^{[L-1]}$ from the previous layer
  - The term $\mathbf{h}^{[L-1]}$ effectively scales the weight update based on how active the previous layer's neurons were.
  - If the activation $\mathbf{h}^{[L-1]}$ is large, then the weight update will also be larger. I.e. the model learns more aggressively based on that input.
  - Conversely, if the activations are small, the updates will be smaller, suggesting that those inputs had less influence on the final output.
  - **Intuition**: This makes sense because if a neuron in the previous layer was highly activated (meaning it contributed significantly to the output), its corresponding weights should be adjusted more significantly based on the error at the output. The update reflects the importance of that neuron in contributing to the error.

Other Layers

For other layers, we work **recursively**:

Error signal at layer l

$$\delta^{[l]} = \left(\mathbf{W}^{[l+1]}\right)^T \delta^{[l+1]} \cdot g'\left(\mathbf{a}^{[l]}\right)$$

Describes how to compute the error signal $\delta^{[l]}$ for any hidden layer l in the neural network.

Illustrates how the error signal is propagated backward through the network, allowing the error from layer $l+1$ to inform the current layer's error.

- $\delta^{[l+1]}$: Represents the error signal from the next layer (layer $l+1$), indicating how much the output of layer $l+1$ contributes to the overall error.
- $\mathbf{W}^{[l+1]}$: The weight matrix for layer $l+1$. The transpose $(\mathbf{W}^{[l+1]})^T$ allows for the back-propagation of the error signal to layer l.
- $g'(\mathbf{a}^{[l]})$: The derivative of the activation function at layer l evaluated at the pre-activation values $\mathbf{a}^{[l]}$, scaling the error signal based on the sensitivity of the activation function.

Thus, this equation shows how the error from the next layer influences the current layer's error, adjusted by the activation function.

The gradient with respect to the weights of layer l ($\mathbf{W}^{[l]}$) is given by:

$$\nabla_{\mathbf{W}^{[l]}} L = \delta^{[l]} \cdot \mathbf{h}^{[l-1]}$$

Shows how the weights in layer l are adjusted (to minimise loss) based on the error signal and the activations of the previous layer.

Emphasizes the influence of those activations in contributing to the output.

- $\nabla_{\mathbf{W}^{[l]}} L$: Denotes the gradient of the loss function L with respect to the weights $\mathbf{W}^{[l]}$ in layer l, **indicating how to adjust the weights to minimize the loss.**
- $\delta^{[l]}$: The error signal computed for layer l, showing how much the output from this layer contributes to the error in the next layer.
- $\mathbf{h}^{[l-1]}$: The activations from the previous layer $l-1$, scaling the weight update based on how active the neurons in that layer were.

Dimensions of the Gradient

$$\nabla_{\mathbf{W}^{[l]}} L \in \mathbb{R}^{(H^{[l]}+1)\times(H^{[l-1]}+1)}$$

$$\delta^{[l]} \in \mathbb{R}^{1\times(H^{[l-1]}+1)}$$

dimensions incorrect??? should be below dims???

$$\delta^{[l]} \in \mathbb{R}^{1\times(H^{[l]}+1)}$$

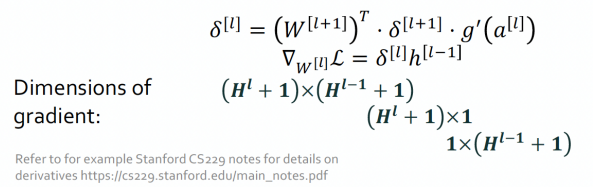


Figure 3.6: Dimensions of the gradient

Gradient with Respect to Weights: The gradient of the loss L with respect to the weights $\mathbf{W}^{[l]}$ in layer l has dimensions given by:

$$\nabla_{\mathbf{W}^{[l]}} L \in \mathbb{R}^{(H^{[l]}+1)\times(H^{[l-1]}+1)}$$

- $H^{[l]}$: The number of neurons (units) in the current layer l.

- $H^{[l-1]}$: The number of neurons in the previous layer $l-1$.

- The $+1$ terms account for the inclusion of bias terms in both layers. This means:

  - Each column corresponds to a neuron in layer $l-1$ (including the bias unit).

  - Each row corresponds to a neuron in layer l (including the bias unit).

Thus, the gradient matrix $\nabla_{\mathbf{W}^{[l]}} L$ represents how the weights connecting layer $l-1$ to layer l should be adjusted based on the error signals.

Error Signal: The error signal $\delta^{[l]}$ for layer l has dimensions given by:

$$\delta^{[l]} \in \mathbb{R}^{1\times(H^{[l-1]}+1)}$$

- This dimension indicates that $\delta^{[l]}$ is a row vector.

- The dimension $H^{[l-1]}+1$ accounts for the number of neurons in the previous layer (including the bias unit).

- This error signal captures how much each neuron in the previous layer contributed to the error at the current layer.

The dimensions of the gradients and error signals ensure that the matrix operations during backpropagation are compatible.

3.5 Minibatch Stochastic Gradient Descent

3.5.1 Stochastic Gradient Descent

SGD updates model parameters iteratively using a single data point per step, resulting in *many updates* and *noise-prone* convergence.

Batch Gradient Descent processes the *entire dataset* at once, reducing noise and requiring fewer updates. However, this approach is *memory-intensive*, as it requires storing large matrices of size $(H + 1) \times d$, where H is the number of hidden units, and d is the dimensionality of the data, potentially causing memory overflow.

1. **Gradient Matrices:** These contain the gradients (partial derivatives) of the loss function with respect to each parameter in the network for the entire dataset. For a neural network layer with H hidden units and d-dimensional input data, the gradient matrix for that layer has dimensions $H \times d$. For the entire network, the storage requirements sum across layers, often resulting in matrices with dimensions $(H + 1) \times d$, where $H + 1$ accounts for weights across layers plus biases.
2. **Activation Matrices:** These matrices store the activations (outputs) from each layer of the network for all data points in the batch, especially during backpropagation, where the gradients for deeper layers depend on previous layers' activations. For a batch of N samples and a layer with H units, the activation matrix for that layer has dimensions $N \times H$.
3. **Parameter Matrices:** These matrices hold the network parameters (weights and biases) for each layer, which are updated based on the gradients calculated from the entire batch. Each parameter matrix's size depends on the architecture of the layer it connects.

Minibatch Gradient Descent offers a balance: it processes small, manageable subsets (minibatches) of the data, reducing memory load and computational cost while preserving a more stable convergence pattern.

In stochastic gradient descent (SGD), for every weight update r, the process is as follows:

$$\mathbf{W}^{(r+1)} = \mathbf{W}^{(r)} + \eta^{(r)} \Delta \mathbf{W}^{(r)}$$

Where:

- $\mathbf{W}^{(r)}$ represents the weights before the update.
- $\eta^{(r)}$ is the learning rate for the current update r.
- $\Delta \mathbf{W}^{(r)}$ is the gradient computed for a single datapoint i at iteration r.

Key Characteristics of Stochastic Gradient Descent:

A for loop with one computation at a time, sequentially.

- **Single Datapoint**: In SGD, the gradient is computed for one datapoint at a time, which allows for quicker updates but introduces noise in the optimization process.
- **For-loop**: The weight updates happen in a for-loop, processing one computation at a time sequentially. This can lead to slower convergence compared to batch updates.

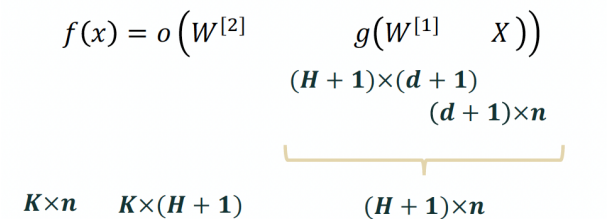


Figure 3.7: batch gradient descent - forward pass dimensions

3.5.2 Batch Gradient Descent

In contrast, batch gradient descent computes the gradient for all data points $i, \ldots, n$ at each weight update r:

$$\mathbf{W}^{(r+1)} = \mathbf{W}^{(r)} + \eta^{(r)} \frac{1}{n} \sum_{i=1}^{n} \Delta \mathbf{W}_i^{(r)}$$

Where:

- The term $\Delta \mathbf{W}_i^{(r)}$ represents the gradients calculated for each datapoint i.
- The gradient is averaged over all n datapoints, leading to a more stable estimate of the gradient direction.

Key Characteristics of Batch Gradient Descent:

- **Parallel Computation**: The weight updates $\Delta \mathbf{W}_i^{(r)}$ for each datapoint per epoch can be computed in parallel, taking advantage of modern computational resources (e.g., GPUs).
- **Efficiency**: Much more training data can be processed per unit of time compared to SGD, but it may also result in higher memory usage and can lead to slower convergence because updates are made less frequently.

Vectorization in Batch Gradient Descent

During the forward pass over all datapoints, the computation can be expressed as:

$$f(\mathbf{x}) = o\left(\mathbf{W}^{[2]} g\left(\mathbf{W}^{[1]} \mathbf{X}\right)\right)$$

Where $\mathbf{X}$ is the input matrix containing **all** training examples, making the operations **efficient** and **leveraging matrix multiplications**.

- $\mathbf{W}^{[2]}$: Weight matrix connecting the hidden layer to the output layer.
- $\mathbf{W}^{[1]}$: Weight matrix connecting the input layer to the hidden layer.

Therefore, in batch gradient descent, we process the *whole* training dataset *before* we update the weights.

Dimensions:

Cautions:

- For large datasets, vectorizing computations may result in memory issues, particularly if the dataset cannot fit into memory.
- Batch gradient descent is generally slower to converge than stochastic gradient descent (SGD) since updates are made only after a complete pass through the dataset.

3.5.3 Introducing Mini-batches

Our training dataset can be represented as follows:

$$\mathbf{X} \in \mathbb{R}^{d \times n} \quad \text{and} \quad \mathbf{Y} \in \mathbb{R}^{1 \times n}$$

Mini-batches are smaller subsets of the dataset of size B. They help in optimizing the training process by splitting the full dataset into manageable chunks.

Formation of Mini-batches

We divide the entire training dataset into m mini-batches of equal size B:

$$\mathbf{X} = [\mathbf{X}^{\{1\}}, \ldots, \mathbf{X}^{\{m\}}]$$

Where:

- Each mini-batch $\mathbf{X}^{\{t\}}$ consists of B datapoints.
- Minibatch t can be represented as:

$$(\mathbf{X}^{\{t\}}, \mathbf{Y}^{\{t\}}) \quad \text{with} \quad \mathbf{X}^{\{t\}} \in \mathbb{R}^{d \times B} \quad \text{and} \quad \mathbf{Y}^{\{t\}} \in \mathbb{R}^{1 \times B}$$

Example: If we have a dataset of 600,000 datapoints and we use a minibatch size of 100, the number of mini-batches m can be calculated as:

$$m = \frac{600,000}{100} = 6,000$$

This means the dataset is divided into 6,000 mini-batches.

Mini-batch Gradient Descent Algorithm

The mini-batch gradient descent process can be summarized in the following steps for each epoch:

1. For every $t = 1, \ldots, m$:
   (a) Perform a **forward pass** to get the predictions based on the current values of the parameters.
   (b) Compute the **loss** using the predictions and the actual values from the mini-batch.
   (c) Compute the **gradient** for the current values of the weights using a **backward pass**.
   (d) **Update** the parameters using:

$$\mathbf{W}^{(r+1)} = \mathbf{W}^{(r)} + \eta^{(r)} \cdot \frac{1}{B} \sum_{i=1}^{B} \Delta \mathbf{W}_i^{(r)}$$

Typical mini-batch size is $B = 32, 64, 128, \ldots$.

Powers of 2 can be more easily processed by CPU/GPU

Advantages of Using Mini-batches

1. **Efficiency**: Mini-batches allow for a more efficient computation of gradients as they can be processed in parallel. This leads to faster convergence compared to using the entire dataset or just one datapoint.
2. **Stability**: By averaging the gradients over a mini-batch, we reduce the variance of the updates, leading to more stable convergence behavior compared to stochastic gradient descent, which uses a single datapoint at a time.
3. **Flexibility**: The mini-batch size B can be tuned to balance between memory efficiency and computational speed, allowing for adaptation to the resources available.
4. **Regularization**: The inherent noise in the mini-batch updates can provide a regularization effect, helping to escape local minima in the loss landscape.

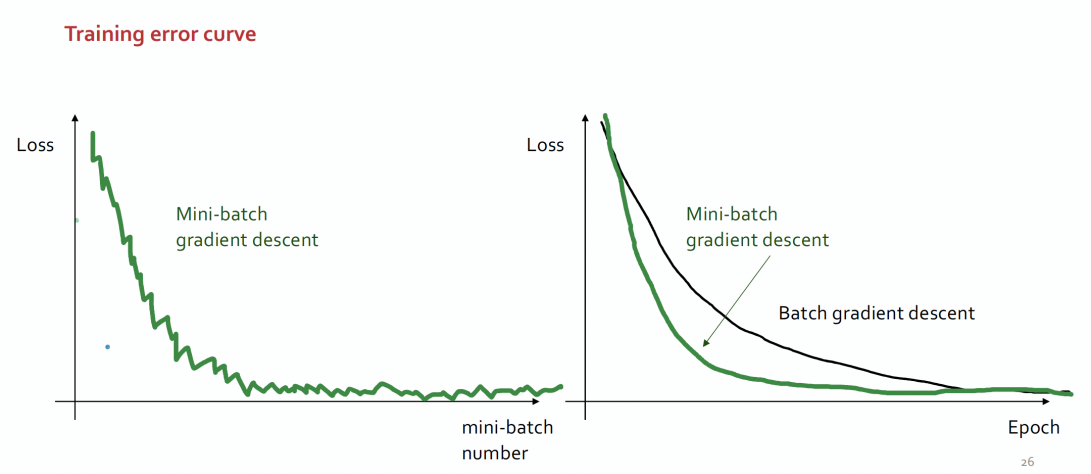


Figure 3.8: Mini-batch gradient descent generally allows for quicker updates compared to batch gradient descent, which processes the entire dataset before making an update. This can lead to faster learning initially. Batch gradient descent typically provides a smoother loss curve with less variance in the updates, making it more stable but potentially slower as the entire dataset must be evaluated before updating the weights. The fluctuations in mini-batch gradient descent can be beneficial as they introduce noise that may help escape local minima, providing a form of implicit regularisation.

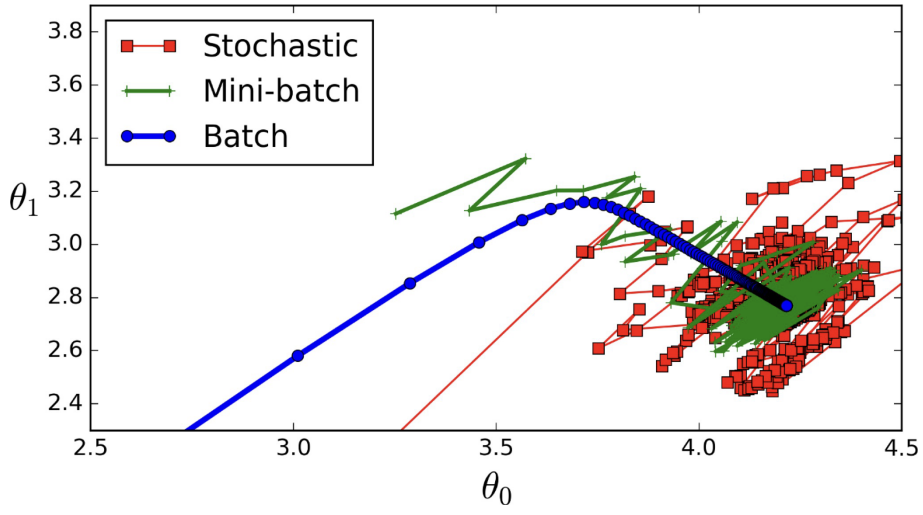


Figure 3.9: Enter Caption

3.6 Training Process

3.6.1 Generalization in Supervised Learning

Supervised Learning Problem Formulation

In supervised learning, we often make the following assumptions:

- The data consists of independent and identically distributed (IID) random variables drawn from a distribution $P(X, Y)$.
- We aim to develop a model that learns from this data and generalizes well to out-of-sample data that is drawn from the same distribution.

> Sometimes, out of sample data come from a different distribution $Q(\cdot) \neq P(\cdot)$. This is distribution shifts.

Training Error

The training error is a measure of how well the model performs on the training data and can be defined mathematically as a **direct function** fo the training data the model:

$$R_{\text{train}}[X, y, f] = \frac{1}{n} \sum_{i=1}^{n} L(x_i, y_i, f(x_i))$$

Where:

- n is the number of training examples.
- L is the loss function measuring the difference between predicted and actual values.

Generalization Error

The true generalization error is defined as the **expectation** taken with respect to the underlying distribution:

$$R[P, f] = \mathbb{E}_{(x,y)\sim P}[L(x, y, f(x))] = \int \int L(x, y, f(x))p(x, y)\, dx\, dy$$

Since we typically do not know the true distribution P, we need to **estimate** the generalization using a test set, computing it similarly to the training error.

Training, Validation, and Test Sets

A typical modeling setup for training deep learning models involves dividing the data into three sets:

- **Training Set:** Used to train the model and adjust its parameters.
- **Validation Set:** Used for hyperparameter tuning and to evaluate the model during training.
- **Test Set:** Remains **locked away** until the end of the experiments, used to assess the final performance of the model.

Figure 3.10: Enter Caption

Cross-Validation: Cross-validation can also be employed for hyperparameter tuning and model selection. While cross-validation is often useful, it can be **computationally intensive in deep learning** scenarios.

Understanding Training Error and Generalization

- A low training error does not guarantee that the model generalizes well. The test error must also be considered for assessing generalization.

- If the training error does not decrease considerably and training and validation errors are similar, the model might be too simple (ie. is unable to learn the pattern from the data), resulting in underfitting.

- If the training error is significantly lower than the validation error, it indicates overfitting to the training data.

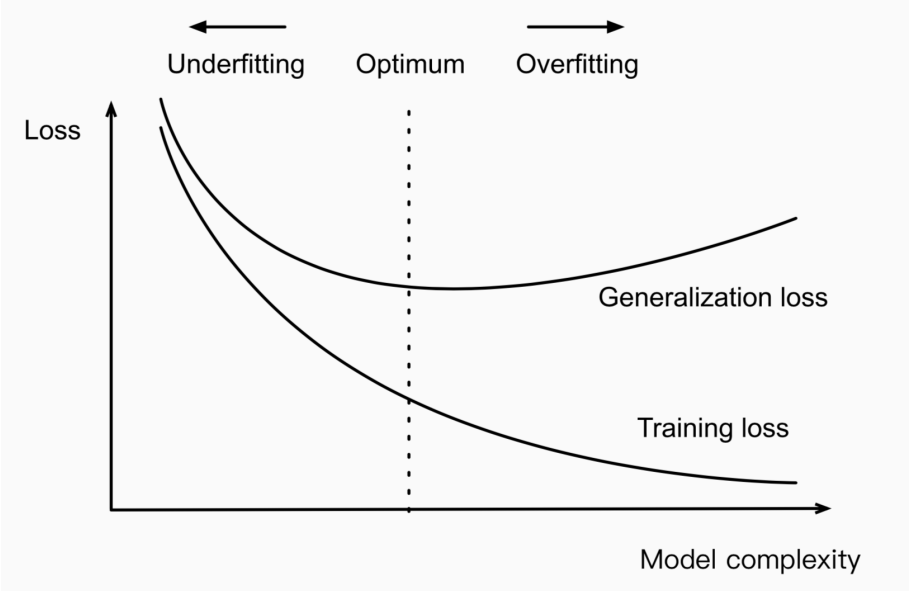


Figure 3.11: Enter Caption

Often we use **early stopping** in DL (rather than cross validation).

1. As cross validation is too *computationally intensive*.

2. As we are so *over-parameterised from the start* in DL, we think in terms of epochs rather than complexity of a model class (it is given that we are complex; we are not about to reduce complexity).

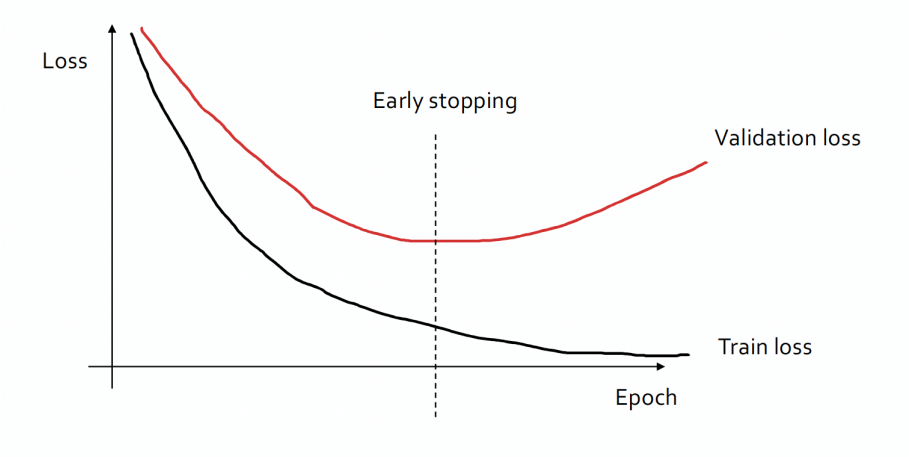


Figure 3.12: Enter Caption

3.6.2 Performance Metrics Common in Deep Learning

Binary Classification Metrics

- **Accuracy:**

$$\text{Accuracy} = \frac{\text{correct classifications}}{\text{all classifications}} = \frac{\sum_{k=1}^{K} TP_k + TN_k}{K}$$

Accuracy can be misleading when dealing with imbalanced datasets, where one class significantly outweighs the other.

- **Precision:** *(thoroughness wrt model's positive predictions)*

$$\text{Precision} = \frac{\text{correct positive classifications}}{\text{all positive classifications}} = \frac{TP}{TP + FP}$$

Precision measures the accuracy of *positive* predictions, emphasizing the relevance of the predicted positives.

- **Recall:** *(thoroughness wrt data itself)*

$$\text{Recall} = \frac{\text{correct positive classifications}}{\text{all true positives}} = \frac{TP}{TP + FN}$$

Recall indicates the model's ability to identify all relevant instances, focusing on the actual positives.

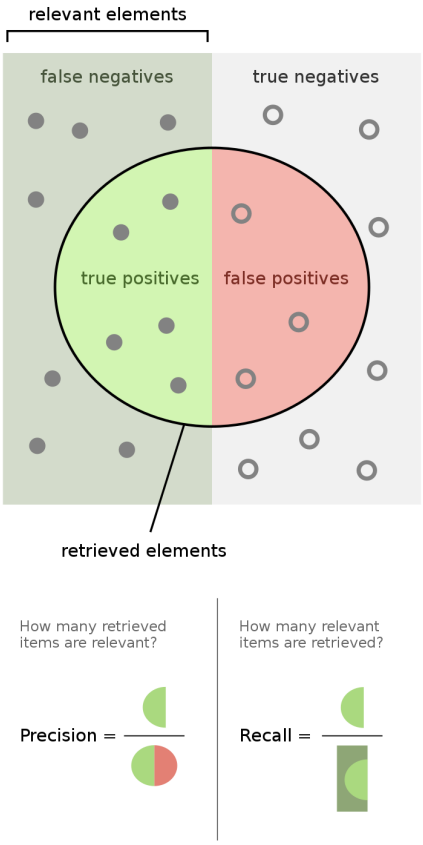


Figure 3.13:

- **F-score:** To combine precision and recall into a single metric, the F-score is computed as follows:

$$F_\beta = \frac{(\beta^2 + 1) \cdot \text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}}$$

Here, $\beta > 1$ emphasizes recall, while $\beta = 1$ results in the F1-score, balancing precision and recall.

We cab fine tune the F-score's weighting towards prevision vs recall depending on the task at hand.

$$F_\beta = \frac{(\beta^2 + 1) \cdot TP}{(\beta^2 + 1) \cdot TP + FP + FN}$$

F1 Score: where $\beta = 1$

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

$$F_1 = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

ROC AUC (Area Under the ROC Curve)

Particularly useful when dealing with imbalanced datasets, where traditional accuracy may not be sufficient to evaluate the model's performance.

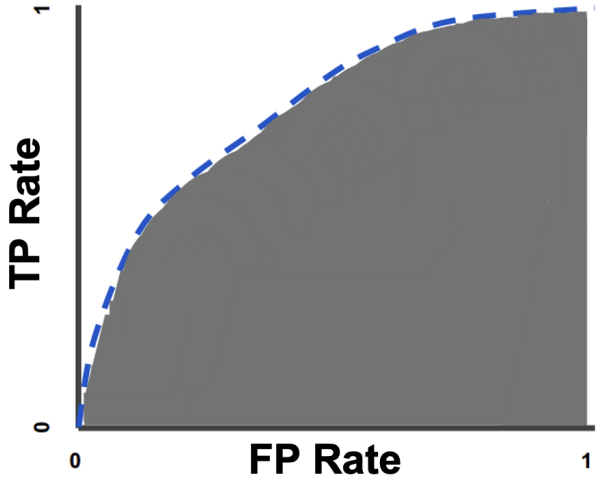


Figure 3.14: blue line: trade off across different decision thresholds

The ROC curve

A graphical representation of the trade off between true positive rate against the false positive rate across different thresholds.

True Positive Rate (TPR):

$$TPR = \frac{\text{True Pos}}{\text{All Pos}} = \frac{TP}{TP + FN}$$

False Positive Rate (FPR):

$$FPR = \frac{\text{False Pos}}{\text{All Neg}} = \frac{FP}{FP + TN}$$

As the threshold for classifying a sample as class 1 is varied (from 0 to 1), the model will generate different TPR and FPR values, which define the ROC curve.

- Top-left corner (0, 1): Ideal performance where TPR is 1 (all positives correctly identified) and FPR is 0 (no negatives are misclassified as positives).
- Diagonal line (from (0, 0) to (1, 1)): This represents the performance of a random classifier. A model whose ROC curve lies closer to the top-left corner indicates better performance.

AUC (Area Under Curve):

The AUC quantifies the *overall ability* of the model to discriminate between positive and negative classes, being *scale-invariant* and *classification-threshold invariant*.

- **Scale-invariant**. It measures how well predictions are ranked, rather than their absolute values.
- **Classification-threshold-invariant**. It allows to evaluate the model's performance by considering all classification thresholds simultaneously. – **But** sometimes we want intuitive thresholds (e.g. 0.5)

E.g.

- AUC = 1: Perfect classifier. The model makes no errors; it perfectly separates the positive and negative classes.
- AUC = 0.5: The model performs no better than random guessing. The ROC curve lies along the diagonal line, and the classifier cannot distinguish between the positive and negative classes.
- AUC $<$ 0.5: This indicates a model that is worse than random, as it consistently misclassifies the classes.

Multi-Class Classification Metrics

In multi-class settings, precision and recall can be extended through:

- **Macro-Averaging:**

$$\text{Precision}_M = \frac{\sum_{k=1}^{K} TP_k}{\sum_{k=1}^{K} (TP_k + FP_k)} \cdot \frac{1}{K}$$

  Macro-averaging treats all classes equally...

- **Micro-Averaging:**

$$\text{Precision}_\mu = \frac{\sum_{k=1}^{K} TP_k}{\sum_{k=1}^{K} TP_k + FP_k}$$

  ...While micro-averaging favours larger classes.

  The **average F-score** can similarly be computed:

$$F_{\text{score}} = \frac{(\beta^2 + 1) \cdot \text{precision}_M \cdot \text{recall}_M}{\beta^2 \cdot \text{precision}_M + \text{recall}_M}$$

3.6.3 Training Tips

Underfitting

- **Problem**: Training error does not decrease (we are stuck in a local optima).
- **Solutions**:
  - Increase model complexity or change the model type.
  - Improve optimization techniques:
    - Use momentum in gradient descent.
    - Apply batch normalization.
    - Adjust the learning rate (increase or use adaptive learning rate).
    - Switch to ReLU activation to avoid vanishing gradients.
    - Properly initialize weights to prevent saturation of activation functions.
  - Debug your code to find potential issues.

Overfitting

- **Problem**: Training error is very low, but validation performance is poor.
- **Solutions**:
  - Employ regularization techniques to reduce model complexity.
    - Use weight sharing.
    - Implement dropout.
    - Apply weight decay.
    - Encourage sparsity in hidden units.

Visualizing Features (parameters)

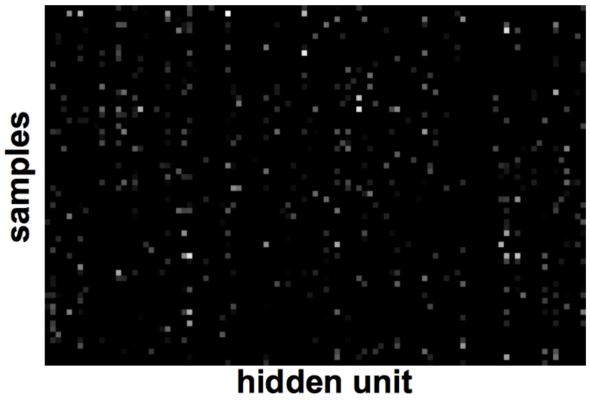


Figure 3.15: Good training will show sparse hidden units across samples, indicating effective feature extraction.

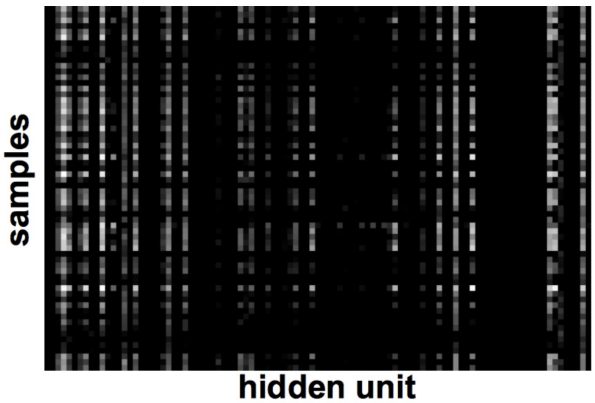


Figure 3.16: Poor training may result in hidden units exhibiting strong correlations and ignoring input, showing less structured patterns.

Training Tips: Common Issues

- If the training error diverges:

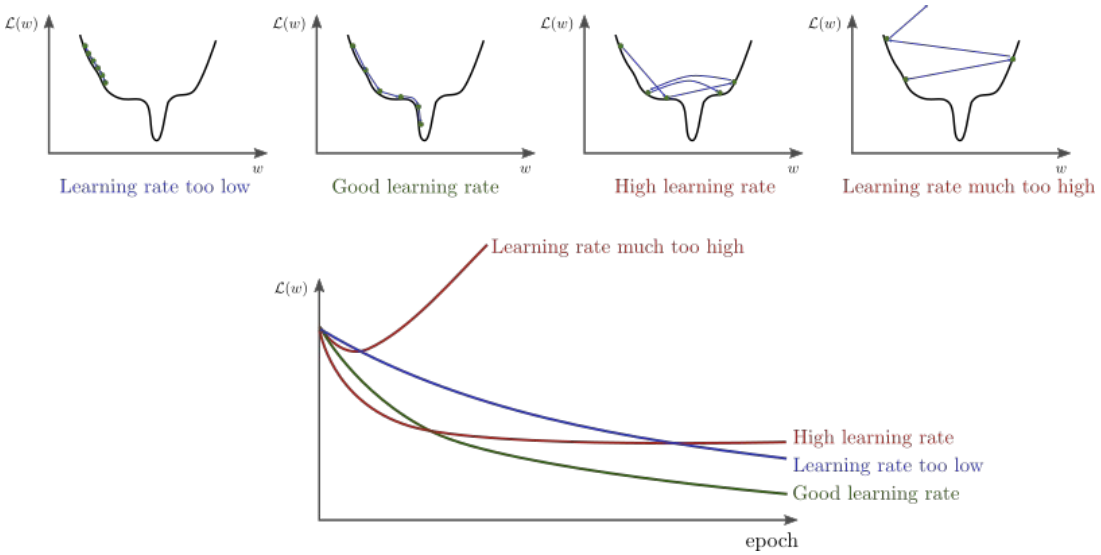


Figure 3.17: Enter Caption

  - The learning rate may be too high; consider decreasing it.
  - Check for bugs in the backpropagation implementation.

- If loss is minimized but accuracy remains low, evaluate the appropriateness of the loss function for the specific task.

3.7 Vanishing Gradient Problem

3.7.1 Saturation

Sigmoid Activation Function

In a two-layer neural network, the output function $f_k(X)$ can be expressed as:

$$f_k(X) = o\left(b_k^{[3]} + \sum_{l=1}^{H^{[2]}} w_{kl}^{[3]} \sigma\left(b_l^{[2]} + \sum_{i=1}^{H^{[1]}} w_{li}^{[2]} \sigma\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right)\right)\right)$$

The component of the gradient with respect to the weights $w_{ij}^{[1]}$ can be **generally** expressed as:

$$\frac{\partial L(f(X), y)}{\partial w_{ij}^{[1]}} = \sum_{k=1}^{K} \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial a_k^{[3]}} \cdot \sum_{l=1}^{H^{[2]}} \frac{\partial a_k^{[3]}}{\partial h_l^{[2]}} \cdot \frac{\partial h_l^{[2]}}{\partial a_l^{[2]}} \cdot \sum_{i=1}^{H^{[1]}} \frac{\partial a_l^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

It consists of a series of partial derivatives that propagate back through the layers of the network. In this form, the activations and their derivatives are not yet specified in terms of a particular activation function.

Here we highlight the components that refer to activation function and its derivative:

$$\frac{\partial L(f(X), y)}{\partial w_{ij}^{[1]}} = \sum_{k=1}^{K} \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial a_k^{[3]}} \cdot \sum_{l=1}^{H^{[2]}} \frac{\partial a_k^{[3]}}{\partial h_l^{[2]}} \cdot \frac{\partial h_l^{[2]}}{\partial a_l^{[2]}} \cdot \sum_{i=1}^{H^{[1]}} \frac{\partial a_l^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

If we introduce the specific form of the **sigmoid activation function** $\sigma(a) = \frac{1}{1+e^{-a}}$, and its derivative $\sigma(a)(1-\sigma(a))$, then we get:

$$= \sum_{k=1}^{K} \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial a_k^{[3]}} \cdot \sum_{l=1}^{H^{[2]}} \frac{\partial a_k^{[3]}}{\partial h_l^{[2]}} \cdot \sigma(a_l^{[2]})(1 - \sigma(a_l^{[2]})) \cdot \sum_{i=1}^{H^{[1]}} \frac{\partial a_l^{[2]}}{\partial h_i^{[1]}} \cdot \sigma(a_i^{[1]})(1 - \sigma(a_i^{[1]})) \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}$$

This use of the **sigmoid activation function leads to the vanishing gradient issue**. The *derivative of the sigmoid function is small when the activation is close to 0 or 1*, which contributes to the vanishing gradient problem.

This can be seen visually:

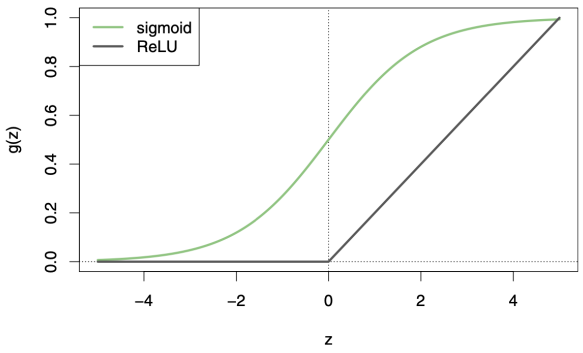


Figure 3.18: at the extremes, the gradient becomes flat - i.e. close to 0, NB: ReLu dos not suffer this.

Saturation of Sigmoid Function

The sigmoid activation function can saturate when its output $\sigma(a)$ approaches either 0 or 1. In **both** these scenarios, the derivative $\sigma(a)(1 - \sigma(a))$ approaches 0 (it's flat). Therefore:

- If a neuron is "firing" (i.e., $\sigma(a)$ is close to 1), the output of the neuron becomes less sensitive to changes in the input.
- If a neuron is "not activated" (i.e., $\sigma(a)$ is close to 0), it also becomes less sensitive to input variations.

This results in very small gradients for weights during backpropagation:

$$\frac{d\sigma(a)}{da} = \sigma(a)(1 - \sigma(a)) \quad \text{is very small.}$$

Problems arise when we multiply many of the sigmoid activation functions with each other...

Multiplication of Small Gradients

In deep networks, we multiply multiple small values (gradients) during backpropagation, leading to an infinitesimal gradient for networks with many layers.

This cumulative multiplication causes the gradients to vanish, preventing effective learning for deeper networks.

This phenomenon is known as the **vanishing gradient problem**, where the gradient becomes so small that the weights do not change significantly during the update step, hindering the learning process.

This saturation effect is also observed with the hyperbolic tangent (tanh) activation function, albeit to a lesser degree.

3.7.2 Overcoming the Vanishing Gradient Problem

Solution 1. Other activation functions

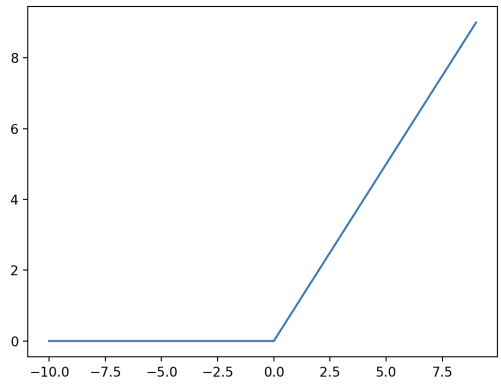


Figure 3.19: ReLU

The Rectified Linear Unit (ReLU) activation function is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

Or:

$$g(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

ReLU is a non-saturating activation function, meaning that it does not suffer from the vanishing gradient problem.

ReLU overcomes this by keeping the gradient constant for positive inputs. This allows efficient backpropagation and prevents gradient decay, which accelerates convergence in deep networks.

Moreover, the ReLU's computational efficiency also makes it widely adopted in deep learning. The gradient of ReLU with respect to x is given by:

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

(In practice, the gradient is typically treated as 0 when $x = 0$, with 1 for $x > 0$ and 0 for $x < 0$.)

This gradient of 1 for non-zero values makes it a very quick non-linearity to compute.

Solution 2. Batch Normalization: Keeping Activation in Non-Saturating Regime

Batch normalization (BN) helps maintain activations **within a useful range** by normalizing the pre-activation values to have a mean of zero and a standard deviation of one over each mini-batch, before the activation function is applied. This prevents saturation of activation functions

like Sigmoid and Tanh.

The main steps of batch normalization are:

- Each mini-batch is normalized using the sample mean and variance from that mini-batch.
  $\rightarrow$ Normalised to zero mean and unit variance
- A simplified view of BN for a fully connected layer with input X is given by:

$$h = g(\text{BN}(b + Wx))$$

In batch normalization, two additional learnable parameters (*scale* and *shift*) are introduced to allow the network to learn the optimal distribution. These are learned from the data during training.

Solution 3. Residual Networks: Overcoming the Vanishing Gradient Problem with Skip Connections

Residual networks (ResNets) were introduced to overcome the vanishing gradient problem by introducing skip connections (also known as shortcut connections).

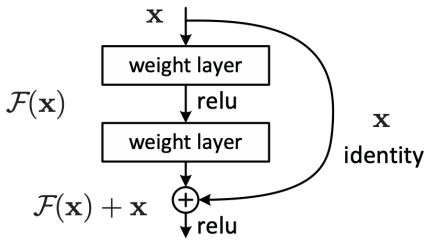


Figure 2. Residual learning: a building block.

These allow the gradient to bypass certain layers during backpropagation. This ensures that gradients do not become too small as they propagate through many layers.

There are two forms of residual connections:

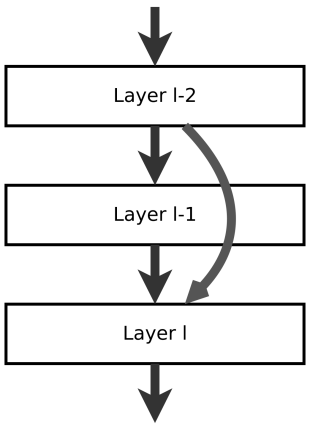


Figure 3.20: Skipping 1 layer

- **Skipping one layer:**

$$h^l = g(W^{[l]} \cdot h^{[l-1]}) + W^{[l-1]} \cdot h^{[l-2]}$$

- **Skipping two layers:**

$$h^l = g(W^{[l]} \cdot h^{[l-1]}) + W^{[l-2]} \cdot h^{[l-3]}$$

This residual learning method helps preserve gradient flow and enables training of very deep networks, which would otherwise suffer from vanishing gradients.

Residual Networks: Implementation in Computer Vision Models

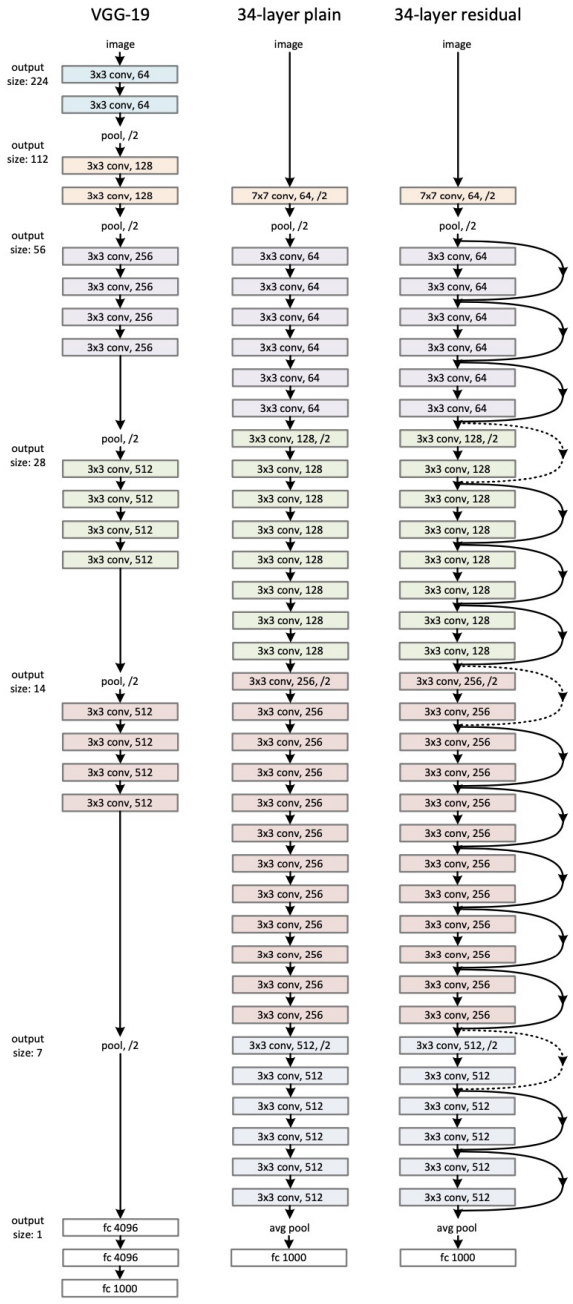


Figure 3.21: Architectural differences between a standard convolutional network (like VGG-19), a plain 34-layer network, and a 34-layer residual network.

- **VGG-19** has a straightforward architecture with no skip connections and approximately 19.6 billion FLOPs (floating-point operations per second).
- **34-layer plain network** attempts to go deeper, but without residual connections, it suffers from the vanishing gradient problem, resulting in lower performance.
- **34-layer residual network** with skip connections solves the gradient decay issue and is more computationally efficient at 3.6 billion FLOPs.

The residual network maintains the benefits of depth without the usual drawbacks, resulting in better performance in tasks such as image classification. The dotted lines in the diagram represent shortcut connections that increase dimensions when needed.

Chapter 4

Convolutional Neural Networks I

Computer Vision Tasks

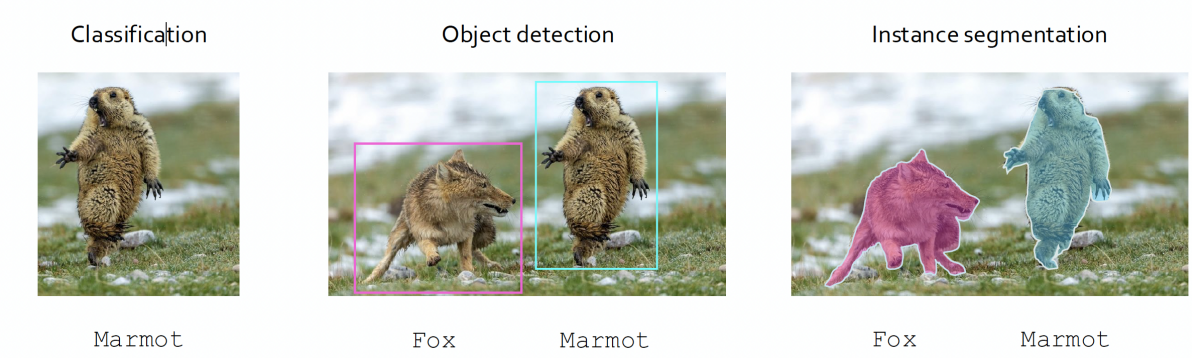


Figure 4.1: Computer Vision

Human vs Computer Image Analysis

Humans

- Look for local features
- Ignore irrelevant info

Computer

- Matrix of pixels
- Light intensity of pixel: numerical value (typically between 0 - 255)
- Color channels
  - Typically 3 channels (RGB)
  - Can be hyperspectral (100s of channels)

4.1 Challenges Solved by Convolutional Layers

CNNs Motivation:

1. Reduce parameters.
2. Leverage local connectivity.
3. Deal with translation invariance & equivalence (treats results same local result irrespective of where it is).

4.1.1 Context: Fully Connected Layers

So far we only worked with "fully connected" layers, where every node is connected to every node of the next layer.

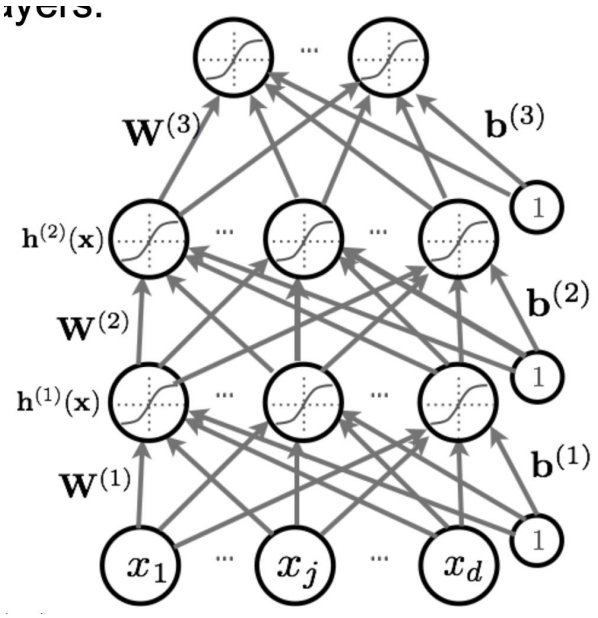


Figure 4.2: Fully Connected network

Mathematically, we represent the activation of a neuron in the next layer as:

$$h_j^{[l]} = \sigma\left(\sum_{i=1}^{H^{[l-1]}} W_{ij}^{[l]} h_i^{[l-1]} + b_j^{[l]}\right)$$

should this be a?
Here:

- $h_i^{[l-1]}$ represents the activations from the previous layer $(l-1)$,
- $W_{ij}^{[l]}$ is the weight connecting neuron i in layer $l-1$ to neuron j in layer l,
  - NB: summation over i: summing over weighted inputs from the prev layer into the next layer node
- $b_j^{[l]}$ is the bias term for neuron j in layer l,
- $\sigma(\cdot)$ is the activation function (often ReLU or sigmoid).

4.1.2 Convolutional Layers: Images and input features

For image analysis and other applications, another type of layer has been very successful: **convolutional layers**.

Challenge 1: Spatial structure and local connectivity

***Problem**: Nearby pixels are often related!*

In fully connected layers, the spatial information present in input data like images is not preserved:

- To feed an image into a fully connected network, the image must first be 'flattened': transformed into an array of pixels.
- The result is a vector where each pixel is taken as one feature.
- Every pixel is treated independently, which can lead to inefficiency when dealing with structured data like images.

The result is that each pixel is a separate feature, and this representation discards important information about the spatial relationships (spatial correlations) between nearby pixels.

***Solution:** Convolutional Layers and Local Connectivity*

Convolutional layers preserve the spatial structure of the input. Instead of treating each pixel independently, convolutional layers apply filters (or kernels) across the image to detect patterns like edges, textures, or objects.

In a convolution operation, we slide a small kernel over the input image. For example, given an image $X \in \mathbb{R}^{d_x \times d_y}$ and a kernel $K \in \mathbb{R}^{r \times r}$, the convolution is computed as:

$$(X * K)_{i,j} = \sum_{p=0}^{r-1} \sum_{q=0}^{r-1} x_{i+p,j+q} k_{r-p,r-q}$$

The kernel detects specific features at different locations in the image, such as edges, corners, or textures, regardless of their position. This helps achieve **translation invariance**, meaning the network can recognize patterns regardless of where they appear in the image.

Challenge 2: High-dimensional Image Inputs

Problem: we want high performance & less parameters!

Images typically contain thousands or millions of pixels, leading to very high-dimensional input spaces. For example:

$$\text{Image size} = 28 \times 28 = 784 \text{ pixels for a small grayscale image.}$$

When we work with images, like a (small) 1 MB image:

$$\text{Image size} = 10^6 \text{ pixels.}$$

If we were to fully connect such a large input image to a hidden layer with $1000(10^3)$ units, the number of parameters we would need to train is:

$$10^6 \times 10^3 = 10^9 \text{ parameters.}$$

This huge number of parameters makes training challenging, requiring a lot of **computational power** and leading to possible **overfitting**. Additionally, this method does not exploit the spatial structure of images, where nearby pixels are often related.

Solution: *Convolutional layers share weights & Pooling Layers reduce dimensionality*

Challenge 3: Translation Invariance and Equivariance

Problem: We need a new type of layer to work as such a ***"detector"***

In image processing tasks, when detecting objects, we care about the *presence* of the object but not their exact position. Convolutional layers are designed to be translation invariant, meaning they focus on detecting patterns regardless of where they appear in the input.

Solution: *kernel filters*

In a convolutional layer, the filter acts as a feature detector (e.g., helmet detector) that slides over the entire image to find areas that match.

Furthermore, CNNs also exhibit **equivariance**, meaning that if the input shifts, the output also shifts by the same amount. This makes them especially useful for tasks like object detection.

Example: Cat Image Feature Detection

Consider the task of classifying an image of a cat. Different filters can be used to detect different parts of the cat, such as its eyes or nose. For example, one filter might activate when it detects an eye, and another when it detects the nose. These activations help the network recognize that the image contains a cat, even if the cat's position in the image changes.

In this case, different parts of the image correspond to different filters, and the combination of these feature detectors enables the network to classify the image accurately.

4.2 Properties of CNNs

CNNs are widely used in Computer Vision. Also, competitive in exploiting one-dimensional sequence structure in time series analysis, audio and text. This is because of:

Local Connectivity

Convolutional neural networks (CNNs) contain convolutional layers that leverage dependencies of nearby features.

This property allows CNNs to exploit the spatial relationships within images more efficiently than fully connected networks.

Translation Invariance and Equivariance.

CNNs exhibit **translation invariance**, which means that the exact position of features within the image is less important. For example, whether an object like a cat is on the left or right of an image, a CNN can still detect it effectively. This is achieved through the sliding window mechanism of convolutional layers, where filters (or kernels) are applied across the image at every position.

CNNs also have **equivariance**, meaning that a shift in the input results in a corresponding shift in the output. If the features in the input image shift by some pixels, the output of the convolutional layer will shift accordingly, preserving the structure of the features.

Locality

CNNs focus on **locality** in early layers by analyzing small regions of the input image (local features like edges, corners). As you move deeper into the network, the layers begin to combine local features into more global, high-level features (such as objects). This hierarchical approach allows CNNs to detect complex structures in images by combining simpler ones.

Invariance to Illumination:

CNNs can also handle variations in illumination. Because they focus on extracting patterns and shapes from the image, changes in brightness do not drastically affect their performance. The filters in CNNs adapt to detect features like edges, which remain consistent even under different lighting conditions.

Computational Efficiency

- CNNs require **fewer parameters** compared to fully connected networks. This is because filters are **shared** across the image, meaning that the same weights are applied to different regions of the image. This weight sharing reduces the total number of parameters and makes CNNs easier to train.

- CNNs are highly **parallelizable**. The convolution operation can be performed simultaneously across multiple parts of the image, making CNNs well-suited for **GPU** acceleration, significantly speeding up training.

4.2.1 Versatility

CNNs are not only limited to computer vision tasks. They are also highly effective in **one-dimensional sequence analysis**, such as:

- **Time series analysis**, where the temporal relationships in the data can be captured by treating the sequence similarly to how images are processed.

- **Audio processing**, where sound wave patterns can be analyzed using CNNs in a manner similar to how images are processed.

- **Text analysis**, especially for tasks like sentiment analysis or language modeling, where convolutional layers help detect important sequences of words or characters.

4.3 Discrete Convolution Operations

The purpose of the convolution is to apply a small filter or **kernel** across an image (or any input data) to extract meaningful features.

This is a grid / matrix of learned features.

4.3.1 Definition of Discrete Convolution

Consider an image input $X \in \mathbb{R}^{d_x \times d_y}$, which is a matrix of pixel values, and a kernel $K \in \mathbb{R}^{r \times r}$, which is a smaller matrix representing a filter. The discrete convolution is computed as:

$$(X * K)_{ij} = \sum_{p=0}^{r-1} \sum_{q=0}^{r-1} x_{i+p,j+q} k_{r-p,r-q}$$

In this operation:

- X is the input image (or matrix),
- K is the kernel (or filter), and
- (i, j) are the indices for the top-left corner of the region in X where the filter is currently being applied.

The offsets p (rows) and q (columns) run from 0 to $r - 1$, where r is the size of the kernel. Each element of the kernel is multiplied with the corresponding element of the image at the current location, and the results are summed to give the output value at (i, j).

4.3.2 Example of Discrete Convolution

Let's compute the convolution for the following example, where the kernel size is $r = 2$ and the input matrix X and kernel K are given as:

$$X = \begin{bmatrix} 0 & 80 & 40 \\ 20 & 40 & 0 \\ 0 & 0 & 40 \end{bmatrix}, \quad K = \begin{bmatrix} 0 & 0.25 \\ 0.5 & 1 \end{bmatrix}$$

We use the discrete convolution formula:

$$(X * K)_{ij} = \sum_{p=0}^{r-1} \sum_{q=0}^{r-1} x_{i+p,j+q} k_{r-p,r-q}$$

where $r = 2$, and the offsets p and q run from 0 to 1.

Notice the minus sign in the indices of k: We can think of this as a filter $\tilde{K}$, which is the kernel K with **rows and columns flipped**. This simplifies the operation considerably.

The flipped kernel $\tilde{K}$ is:

$$\tilde{K} = \begin{bmatrix} 1 & 0.5 \\ 0.25 & 0 \end{bmatrix}$$

Now, applying the convolution at position (i, j) on the top-left corner of the image matrix X, we compute:

$$1 \cdot 0 + 0.5 \cdot 80 + 0.25 \cdot 20 + 0 \cdot 40 = 45$$

Thus, the output value at (i, j) is 45.

4.3.3 Purpose

This discrete convolution operation allows the network to detect local patterns (such as edges, textures, etc.) by applying small kernels across the input.

This process is repeated for each position in the image, with the result being a feature map that highlights the areas where the filter detects relevant patterns.

4.4 Discrete Cross Correlation Operation

The term **Convolutional Neural Network (CNN)** originates from the mathematical concept of convolutions.

However, what is actually computed in most CNN implementations is **cross-correlation**, not convolution in the strict mathematical sense.

While convolution requires flipping the kernel before applying it to the input, cross-correlation does not. Despite this difference, the operation is functionally equivalent for the purpose of feature extraction in neural networks, as both capture local dependencies and patterns in the data efficiently.

4.4.1 Using Convolutional Kernels as Weights

In convolutional layers, the filters or **kernels** act as weight matrices for the input data. The role of the kernel is to extract important features from the input, such as edges or textures in an image.

Here's an important consideration regarding how these kernels function as weights:

Order of Weights

For the computer, the order in which the weights are applied does not matter as long as it is consistent throughout the convolution operation. This means that the order of the rows and columns of the kernel can be flipped, and the operation will still yield the same result. This concept leads us to define a flipped kernel $\tilde{K}$ as our **weight matrix**:

$$\tilde{K} = W$$

where W is the set of learnable weights for the layer. Flipping the rows and columns of the kernel simplifies some convolution operations, but it doesn't change the core concept of applying a filter to an image.

Cross-correlation vs Convolution

While the operation in convolutional layers is referred to as "convolution", it is technically a form of **discrete cross-correlation** between the input X and the weight matrix W.

Mathematically, cross-correlation between an input X and kernel W is written as:

$$(X * W)_{ij} = \sum_{p=0}^{r-1} \sum_{q=0}^{r-1} x_{i+p,j+q} w_{p,q}$$

where the kernel is not flipped as it would be in the mathematical definition of convolution.

Activation of Hidden Layers

The output of the convolution or cross-correlation operation forms the activation of our hidden convolutional layer. The result of this operation is an **activation map** that highlights the regions of the input that correspond to the learned features of the kernel.

Non-linear Activation Function

After applying the convolution, we introduce non-linearity into the model by applying a **non-linear activation function** to the resulting activation matrix. For instance, we can apply a sigmoid function to each entry of the activation matrix.

This non-linearity allows the network to model more complex patterns in the data by stacking multiple convolutional layers with activations.

4.5 Effect of Applying a Convolution

We apply a **convolution** to the input matrix X using a kernel K. The convolution operation slides the kernel across the input, performing **element-wise multiplication** and **summing** the results to produce an output matrix $X * K$.

NB: In this example, the kernel is the same as the kernel/weights with the rows and columns flipped

Pixel matrix:

$$X = \begin{bmatrix} 0 & 0 & 255 & 0 & 0 \\ 0 & 0 & 255 & 0 & 0 \\ 0 & 0 & 255 & 0 & 0 \\ 0 & 255 & 0 & 0 & 0 \\ 255 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Kernel matrix: the convolution highlights the areas that are similar to the filter.

$$K = \begin{bmatrix} 0 & 0.5 \\ 0.5 & 0 \end{bmatrix}$$

Resulting matrix $X * K$: contains values that represent the presence of features detected by the kernel.

$$X * K = \begin{bmatrix} 0 & 128 & 128 & 0 \\ 0 & 128 & 128 & 0 \\ 0 & 255 & 0 & 0 \\ 255 & 0 & 0 & 0 \end{bmatrix}$$

4.5.1 Feature Detection

The convolution highlights areas of the input image **that are similar to the pattern represented by the kernel**. For instance, the kernel in this example is detecting changes in intensity along edges, and the output matrix highlights these areas with higher values (e.g., 128 or 255).

4.5.2 Non-linear Activation

Once the convolution is performed, the result can be passed through a non-linear activation function.

By applying this activation function to each entry of the resulting matrix, we can obtain an output where neurons corresponding to strong feature activations (such as edges) are highlighted.

This process allows the network to focus on relevant areas of the image, and successive layers can further refine these features, eventually leading to high-level feature detection.

4.5.3 Example: Change from dark to light

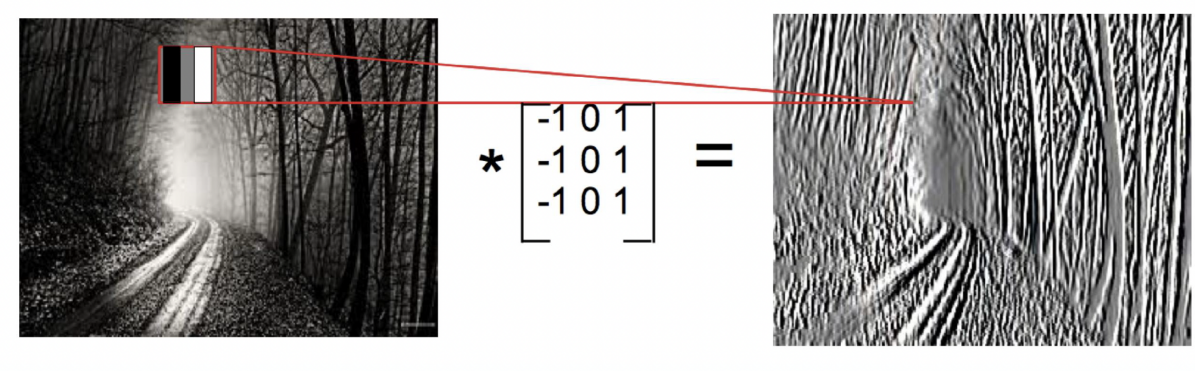


Figure 4.3: NB: this is a correlation operation

4.6 Padding

Loosing Information at the Edges of the Input

When applying a convolutional filter to an image, we encounter a common issue at the edges of the input: the filter may not have enough neighboring pixels to perform a full convolution, which can result in losing information at the borders and shrinking the output size. There are several approaches to handling this problem.

Ignoring the Edges

One simple solution is to ignore the edges, but this leads to the **shrinking** of the image after each convolutional layer.

But: only the pixels where the filter fully fits contribute to the output, and this results in losing valuable information, especially at the borders.

Padding: Adding Extra Pixels

To solve this issue, we can add extra pixels around the borders of the image, ensuring that the convolution can be applied to the entire image, including the edges. This process is called **padding**, and there are several types of padding:

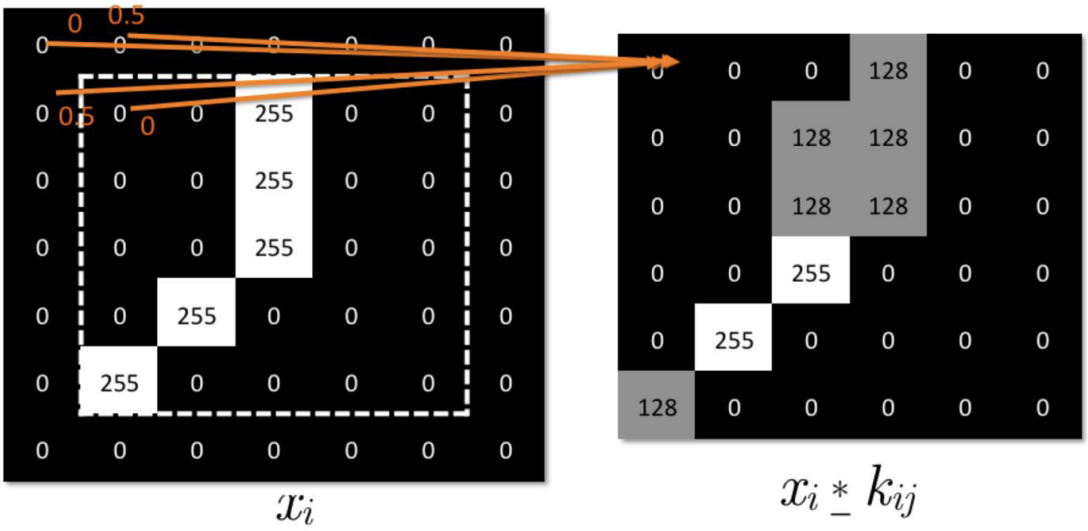


Figure 4.4: Different padding strategies: zero-padding, mirroring, and continuous extension

- **Zero-padding:** The most common method, where extra rows and columns filled with zeros are added around the edges of the image. This ensures that the filter can be applied even to the pixels at the border.
  - This allows the convolutional filter to be applied to the entire image, including the edges, preserving the full image size and avoiding information loss at the borders
  - After applying the convolution, the output image remains the same size as the original; the important edge features are preserved.
- **Mirroring Pixels:** Instead of padding with zeros, we can mirror the pixel values on the border to create the padding. This helps preserve the continuity of the pixel values near the edges.
- **Continuous Extension:** Another option is to assume that the image continues beyond its borders by extending the last pixels indefinitely. This approach is less commonly used, but it is an option in certain scenarios.

Benefits of Zero-padding

The use of padding, particularly **zero-padding**, offers several advantages:

- It **maintains the size** of the input, which is crucial for deep convolutional neural networks where consistent feature map sizes across layers are important.
- It **prevents the loss of important features at the edges** of the image, such as borders and corners, which could be critical for tasks like edge detection.

4.7 Pooling Layers

4.7.1 Motivation: From Local to Global

After applying convolutional layers, we have information for almost every pixel in the image, including where the light gradients and other local features are located. These pixels make up the **feature map**.

However, in many tasks, such as classification, we may not need to know the precise location of each feature. Instead, we may want to aggregate this information to focus on **higher-level patterns or objects** (e.g., identifying whether there is a street in the image).

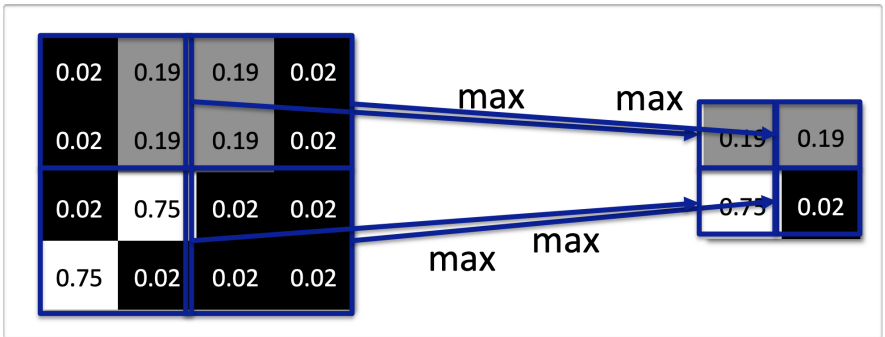
Additionally, we may want to **ignore small local translations**, meaning that the exact position of features is not crucial. To achieve this, we need to "pool" the information by reducing the size of the feature map while preserving the most important details.

4.7.2 Max Pooling Operation

Pooling layers are used to **reduce the dimensionality** of the feature maps while retaining the most important information. One common form of pooling is **max pooling**, where we take the maximum value from each region of the feature map to form a new, smaller pooled feature map.

Example: Max Pooling (with stride = 2)

The window size is 2×2, and the stride is 2. For each window, the maximum value is taken.



As a result, the output feature map is 2×2 instead of 4×4, but it **retains the most important activations from each region**.

4.7.3 Effect of Pooling: Reducing Dimensionality & Local Translation Invariance

Pooling reduces the number of hidden units passed on to subsequent layers and so **reduces the dimensionality** of the feature map significantly, making the neural network more computationally efficient while maintaining important features. For instance, in the example shown, the feature map is reduced from 4×4 to 2×2.

This reduction (also) helps **local translation invariance**: i.e. to generalize the network by making it **less sensitive to the precise location** of features, which is often desirable in tasks like object detection or image classification.

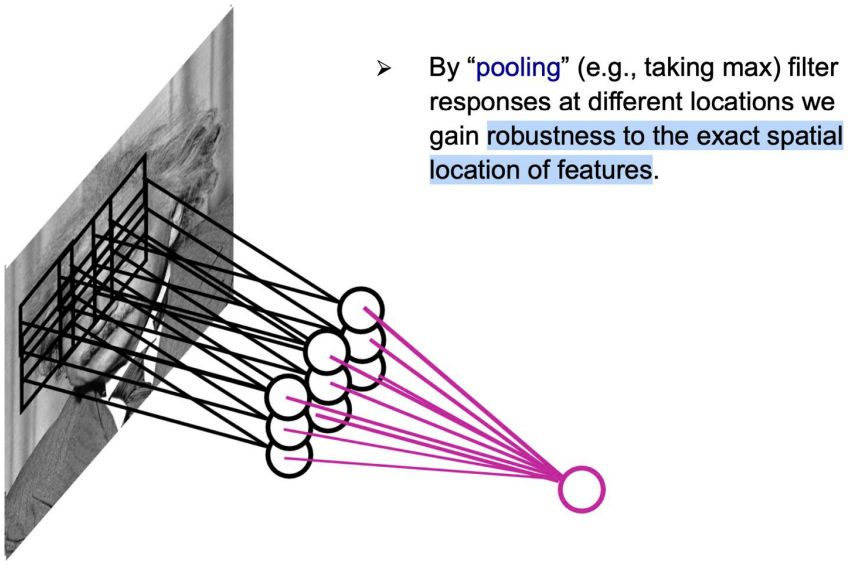


Figure 4.5: translation invariance

Pooling layers introduce a property called **local translation invariance**. This means that small shifts in the position of features in the input image do not significantly affect the outcome after pooling. This is especially useful because it makes the network more robust to slight variations in the location of features.

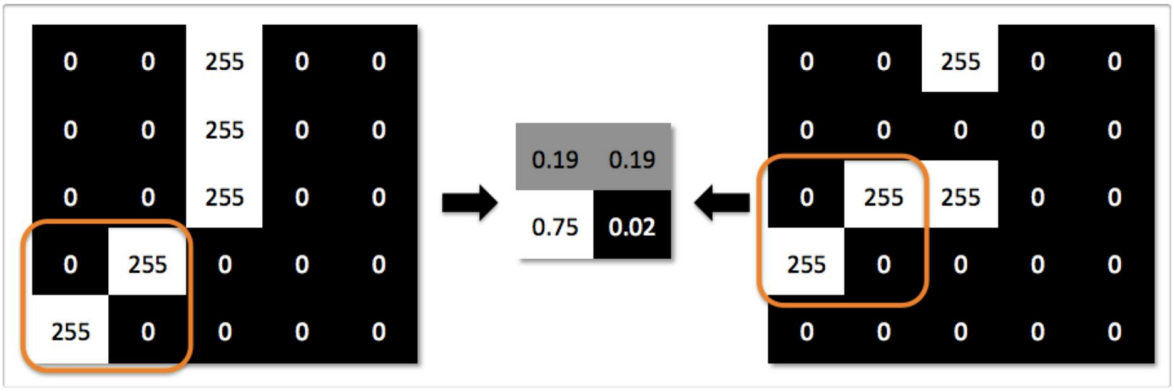


Figure 4.6: In the figure, two slightly different inputs are shown, where the key features (values of 255) are shifted slightly. Despite these shifts, the result of the pooling operation is the same for both inputs. The output feature map remains invariant to these small translations.

This property is often desirable, as we typically do not care about the exact spatial location of features, but rather about whether these features are present.

After applying convolution, sigmoid activation, and pooling, we obtain the same pooled result from both inputs, highlighting the robustness of the pooling layer to local translations.

4.7.4 Mathematical Formulation

Pooling (also called subsampling) is a deterministic operation applied to local, (most often non-overlapping) neighborhoods of hidden units in the feature map. The goal of pooling is to reduce

the dimensionality of the feature map while retaining the most important information.

Overlapping: depends on the **stride**, which defines how far the pooling window moves across the input (i.e., how many pixels the pooling window skips).

Max Pooling

In **max pooling**, we select the maximum value from a local neighborhood in the feature map. Mathematically, for a feature map $h^{[l-1]}$ at layer $l-1$, the max pooling operation can be defined as:

$$h_{jk}^{[l]} = \max_{p,q} h_{j+p,k+q}^{[l-1]}$$

where p and q are the vertical and horizontal indices within the pooling window.

Average Pooling

In **average pooling**, we take the average of the values in the local neighborhood instead of the maximum. The average pooling operation can be written as:

$$h_{jk}^{[l]} = \frac{1}{m^2} \sum_{p,q} h_{j+p,k+q}^{[l-1]}$$

where m is the size of the pooling window, and p and q again run over the pooling window.

Max pooling preserves the most salient features, while average pooling captures an overall representation of the input region.

4.7.5 Pooling and Convolutions

Convolution → sigmoid activation & pooling.

The pooling operation is typically applied after a convolutional layer. The convolution extracts features like edges or textures from the input image, and pooling aggregates these features to focus on more global patterns, reducing sensitivity to exact pixel locations.

Conclusion:

By reducing the number of hidden units passed on to the following layers, pooling **reducing the dimensionality** of feature maps while **retaining the most relevant information**. It introduces invariance to local translations and allows the network to focus on more **abstract patterns** in the data, ultimately improving its ability to classify or detect objects in an image.

4.8 Convolutional Neural Networks

Multiple Input Channels

Color Images and Channels

In convolutional neural networks, images can have multiple **input channels**. Color images, for example, are commonly represented by three channels: red (R), green (G), and blue (B), often

referred to as **RGB**.

Figure 4.7: Color image represented as 3D tensor with RGB channels

This means that the input image is not just a 2D matrix of pixel intensities but a **3D tensor** of size $d_x \times d_y \times 3$, where d_x and d_y are the height and width of the image.

Operation with Multiple Channels

In a CNN, a **single filter** is applied **simultaneously to all input channels**. Each filter has one set of weights per input channel. For example, if the input image has 3 channels (RGB), the filter has 3 corresponding sets of weights.

When the filter is applied:

- It performs a convolution on each input channel separately, using its corresponding weights.
- The results of these convolutions (one for each channel) are then **summed** together to produce a single output value in the output feature map.

This allows the network to combine information from multiple channels (such as the red, green, and blue channels in an image) to **detect patterns that span across all channels**.

For example, given an input with two channels and a filter with two corresponding weight matrices, the convolution operation is:

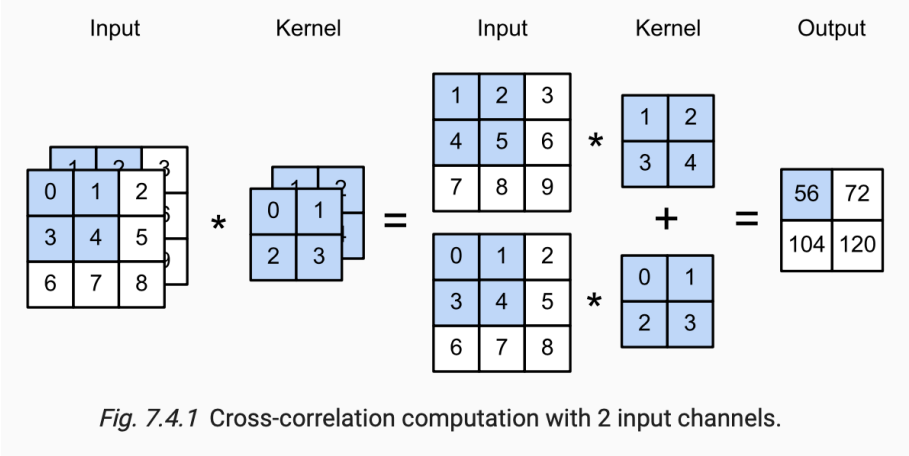


Fig. 7.4.1 Cross-correlation computation with 2 input channels.

Figure 4.8: $(1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 5 \cdot 4) + (0 \cdot 0 + 1 \cdot 1 + 3 \cdot 2 + 4 \cdot 3) = 56$

This operation involves:

- Applying the first kernel to the first channel of the input.

- Applying the second kernel to the second channel of the input.

- Summing the results from both channels.

This shows that a single filter spans across both channels, and the results of each channel's convolution are summed to form the final output.

Multiple Output Channels / Feature Maps

We run multiple filters in CNNs, with each filter capturing different features of the input image.

Each filter produces a **feature map**, and the network can combine these multiple parallel feature maps (layers) to form a final output.

For **multiple input channels** (such as RGB images), each filter in a convolutional layer has a set of weights for each input channel.

- *The filter is applied to all input channels simultaneously, and the results of the convolutions across all channels are summed to produce a single output feature map.*

If we have **multiple filters**, each one operates on all input channels, resulting in a set of output feature maps, one for each filter.

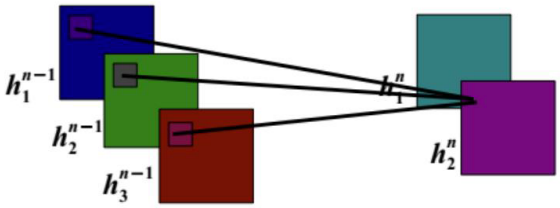


Figure 4.9: **3 input 2 output channels** - NB: 1 filter across multiple input channels to a single feature map... then 2 parallel feature maps as output (given 2 filters)

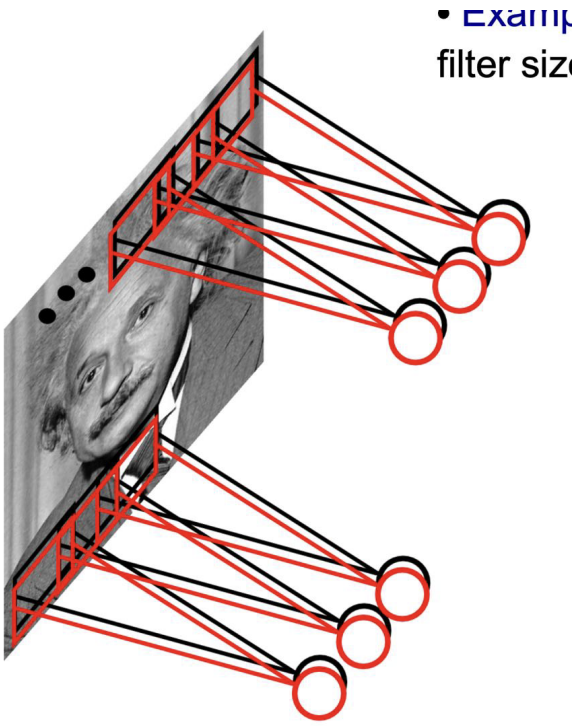


Figure 4.10: 2 Output Channels. Here we run a convolution layer that looks for eyes, then the pooling layer reduces this to 'yes there's an eye' as one data point that is locally invariant and passed onto the network.

Clarification:

For **multiple input channels** (such as RGB images), each filter in a convolutional layer has a set of weights for each input channel.

The filter is applied to all input channels simultaneously, and the results of the convolutions across all channels are summed to produce a single output feature map.

If we have multiple filters, each one operates on all input channels, resulting in a set of output feature maps, one for each filter.

***

- 1-for-1: filter-to-feature map
- A filter is applied simultaneously to all input channels and summed into a single feature map.
- Output can consist of multiple parallel feature maps.

Example: LeNet Architecture

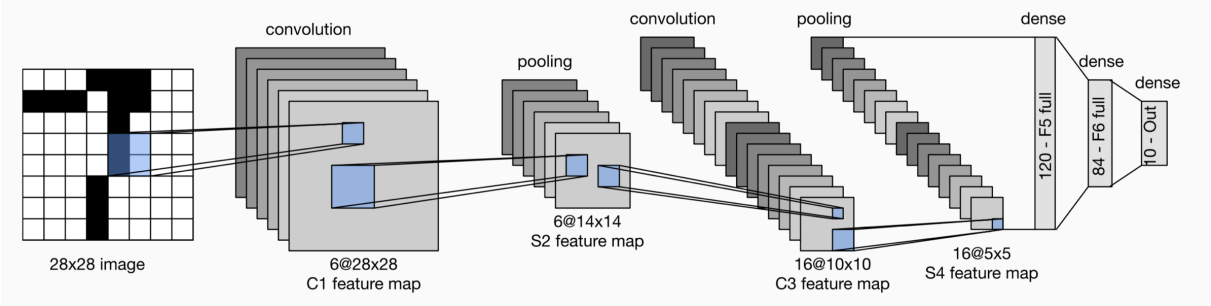


Figure 4.11: LeNet

An example of a CNN is **LeNet**, which uses a 1) **Convolutional encoder** of alternating convolutional layers and pooling layers (LHS) and a **Dense block** of fully connected layers (RHS):

- It uses 6 kernels of size 2×2, resulting in 6 feature maps in layer C_1. (I.e. 6 patterns we are trying to pick up)
- Layer C_1 is a tensor of the dimensions $6 \times 28 \times 28$.
- Pooling layer: keeps the 6 channels in parallel, but reduces spatial dimensionality of each.
- $C3$: now making 16 channels.
- $S4$: uses a 2×2 filter to halve the dimensions.
- Convolutional blocks are flattened before being passed into the dense layers (from 3D image, to 1D array (for classification tasks)).
- *NB: id we had 3 colour channels - they get summed up into the initial feature map right away.*

4.8.1 Training a CNN

For classification, the output layer is a regular, fully connected layer with softmax function.

During training, CNNs are typically trained using **mini-batch stochastic gradient descent.**

The gradients from the output layer are backpropagated through the network, *including through the convolution and pooling layers.*

- For example, in max pooling, the **gradient is only passed to the "winning" unit** (i.e., the unit with the maximum activation), while the **gradients for all other units are zero**.

Feature Visualizations in CNNs

What does a CNN learn?

Convolutional Neural Networks (CNNs) learn hierarchical representations of images. At each layer, the network extracts more complex features:

- In **early layers**, CNNs typically learn low-level features such as **edges** or simple textures.

- In **intermediate layers**, the network captures more abstract patterns such as **textures** and **repeating patterns**.

- In the **higher layers**, CNNs capture increasingly complex structures, like **parts of objects** and even full **objects**.

This progression from simple features (edges) to complex objects reflects how CNNs can understand and classify images by focusing on different scales of abstraction. For example, an early layer might detect a sharp line (edge), while a deeper layer detects a dog's face from a combination of features learned in previous layers.

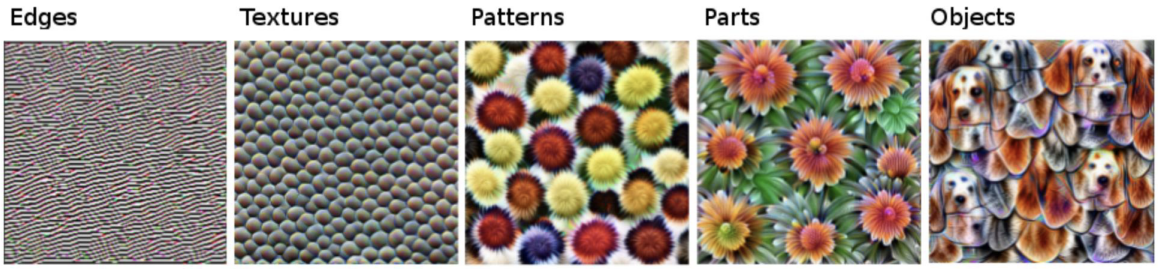


FIGURE 10.1: Features learned by a convolutional neural network (Inception V1) trained on the ImageNet data. The features range from simple features in the lower convolutional layers (left) to more abstract features in the higher convolutional layers (right). Figure from Olah, et al. (2017, CC-BY 4.0) https://distill.pub/2017/feature-visualization/appendix/.

Figure 4.12: Tiling of Neurons in GoogLeNet

How are these features visualized?

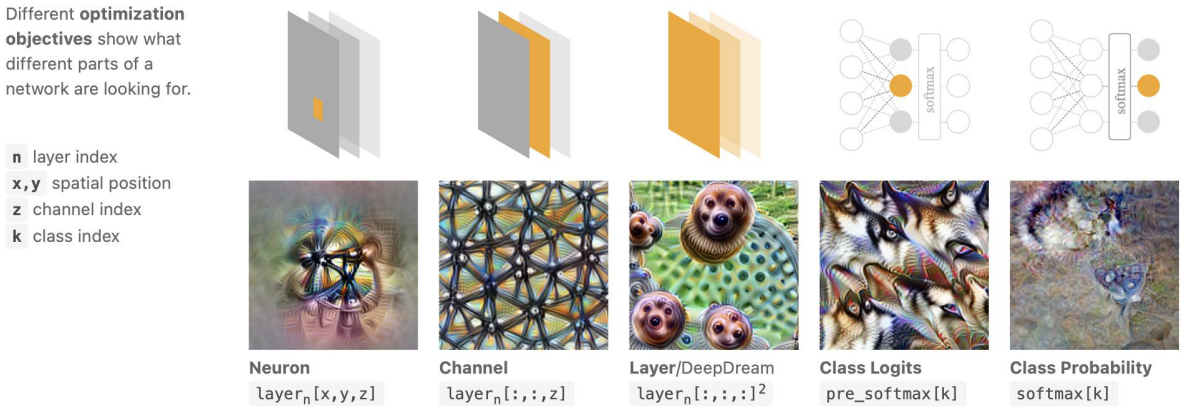


Figure 4.13: https://distill.pub/2017/feature-visualization/appendix/

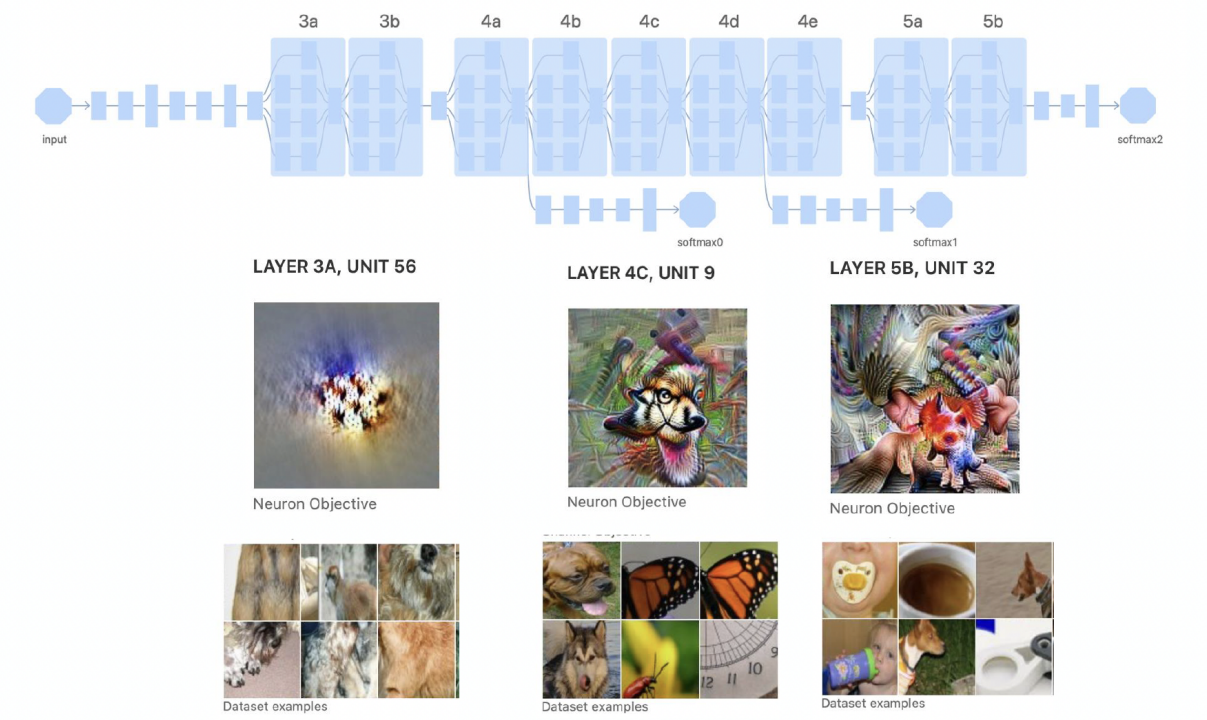


Figure 4.14: https://distill.pub/2017/feature-visualization/appendix/

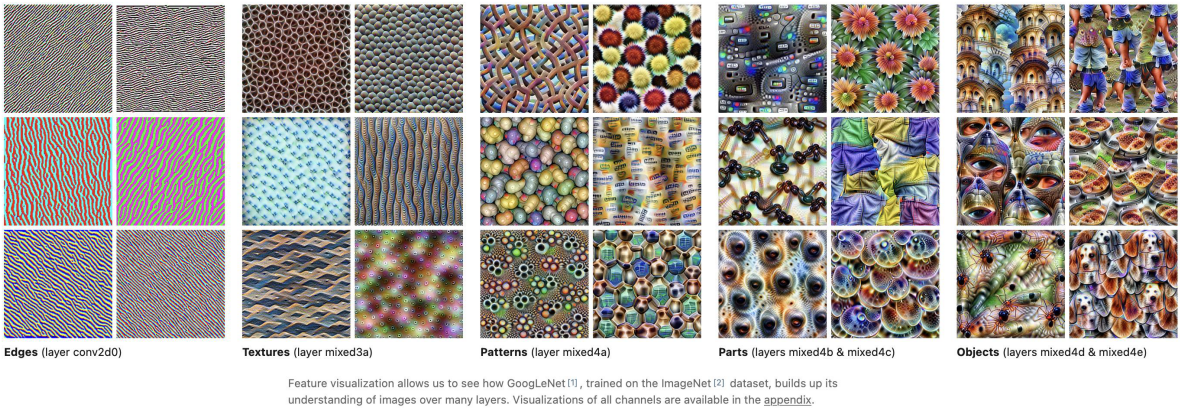


Figure 4.15: From textures to objects

Feature visualization allows us to see what each neuron or layer is learning by generating images that maximize the neuron's activation. Several types of visualizations can be used:

- **Neuron visualization**: We can visualize what input maximally activates a single neuron in a particular layer (represented by layer$[x, y, z]$, where x and y are spatial coordinates, and z is the channel index).

- **Channel visualization**: We can visualize the filters or kernels by optimizing for an entire feature map/channel, giving insight into what patterns that channel detects.

- **Layer visualization (e.g., DeepDream)**: We can perform optimization over entire layers, allowing us to understand what patterns are being captured across multiple neurons within the same layer.

- **Class Logits and Class Probability**: Here, we optimize for specific classes (e.g., maximizing the probability of the class "dog"), showing what features in an image are most strongly associated with that class.

Examples of learned features

- **Edges**: In the very first layers, CNNs learn to detect simple features such as vertical, horizontal, and diagonal edges.
- **Textures**: Deeper layers in the network start detecting repetitive textures. For example, a filter may learn to recognize circular textures or grids.
- **Patterns**: As we move deeper into the network, the filters start recognizing more complex, recurring patterns, like floral patterns or geometric structures.
- **Parts**: Even deeper layers can capture parts of objects (e.g., petals of a flower or fur patterns on an animal).
- **Objects**: In the final layers, CNNs learn full object representations by combining multiple features learned from earlier layers.

Why feature visualization is useful

- Feature visualization provides insight into **what the network is learning** at each layer, helping researchers and engineers understand why a CNN makes certain predictions.
- It helps diagnose **model weaknesses** or biases by revealing what features the network prioritizes.
- Feature visualization also aids in the **interpretability** of deep learning models, which is critical for applications in areas such as medicine, autonomous driving, and other safety-critical fields.

Chapter 5

Convolutional Neural Networks II

Contents

- Labeled data and augmentation
- Modern CNN models
- Object detection
- Semantic Segmentation

5.1 Labeled Data & Augmentation

Leaps in Training Data: State of the art in computer vision driven by large training data

Motivation: Deep NNs only outperform ML models like XGBoost when we have big data regimes.

Historically in CompVision labelled data sets were a constraint (effectively meaning we weren't able to enter big data regimes). So *image labelling has always been a major focus.*

ImageNet and its Significance:

Li FeiFei's ImageNet was the first attempt to address this bottleneck.

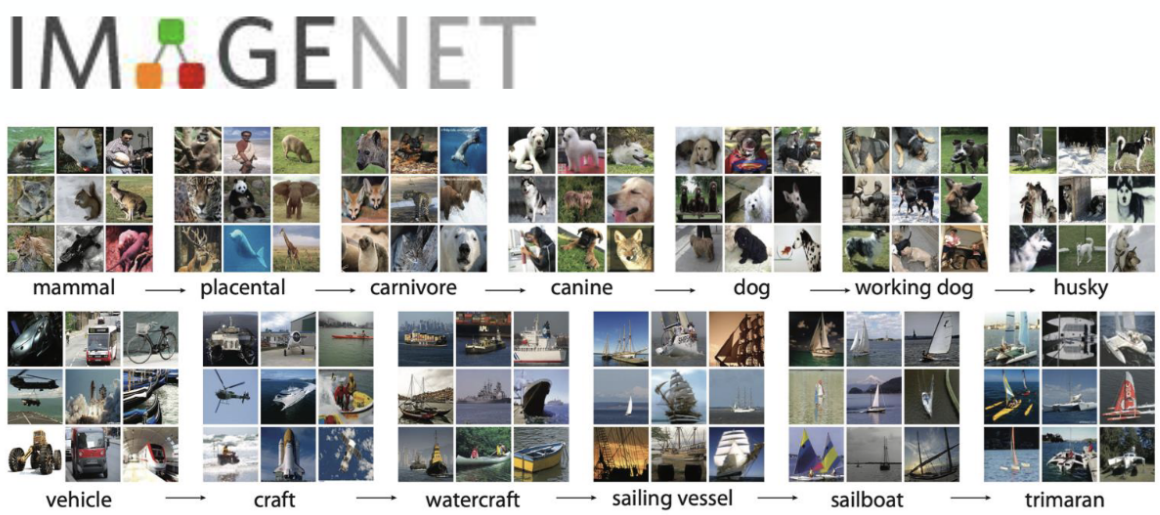


Figure 5.1:

- **ImageNet**: A large-scale dataset that revolutionized computer vision research. Initiated by Li Fei-Fei in 2009, it contains ∼1 million images categorized into 1000 classes. It played a pivotal role in enabling deep learning models, especially Convolutional Neural Networks (CNNs), to reach state-of-the-art performance in image classification tasks.

- **Amazon Mechanical Turk**: Workers on Amazon Mechanical Turk labeled the dataset, allowing large-scale, cost-effective annotation.

- **High Resolution**: Images are relatively high resolution (224x224 pixels), providing sufficient detail for training deep models.

- **Hierarchically Organised**...

- **Comparison to past datasets such as CIFAR-100**: CIFAR-100 is significantly smaller, with only 60,000 images across 100 classes, making ImageNet far more diverse and extensive.

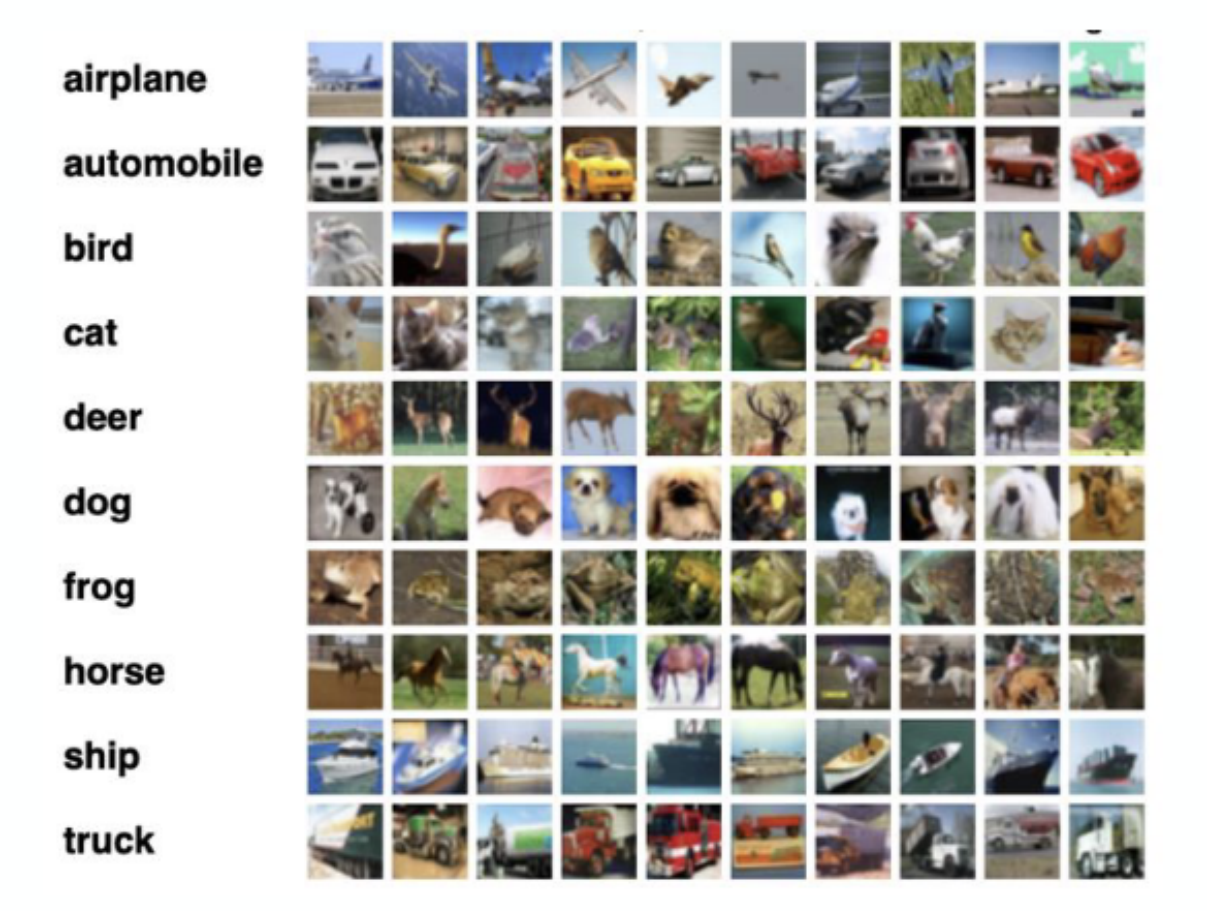


Figure 5.2: CIFAR-10

- **State of the Art: LAION-5B**: Modern datasets like LAION-5B contain billions of images, often paired with additional metadata. This expansion supports more advanced model training approaches, including multimodal learning.

Other Common Datasets

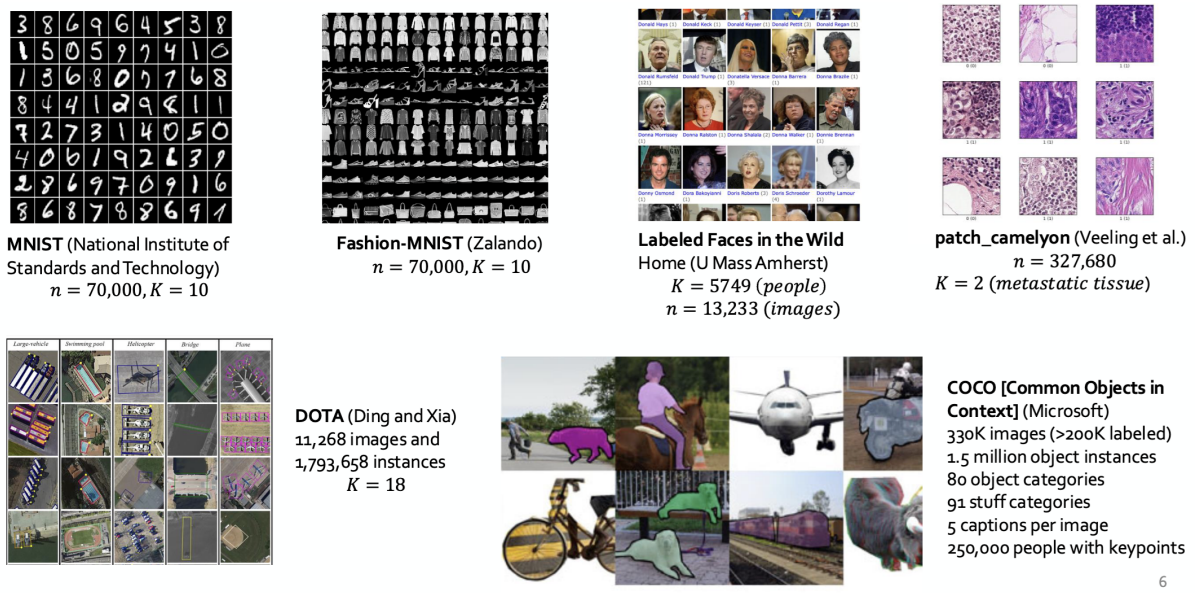


Figure 5.3: Enter Caption

MNIST (National Institute of Standards and Technology):

- This is one of the most well-known datasets in computer vision, consisting of handwritten digits.
- It contains $n = 70,000$ grayscale images of size 28x28 pixels, categorized into $K = 10$ classes (digits 0-9).
- Widely used for benchmarking classification algorithms.

Fashion-MNIST (Zalando):

- Fashion-MNIST is a dataset designed as a drop-in replacement for MNIST, consisting of grayscale images of clothing items.
- It also contains $n = 70,000$ examples, with $K = 10$ classes, including categories like shirts, shoes, and bags.
- The images have the same 28x28 pixel format as MNIST but represent more complex real-world objects.

Labeled Faces in the Wild (U Mass Amherst):

- This dataset focuses on face recognition tasks, containing $n = 13,233$ images of $K = 5,749$ unique people.
- The dataset is widely used for tasks like face verification, face clustering, and related facial recognition tasks.

patch_camelyon (Veeling et al.):

- A medical imaging dataset consisting of histopathologic scans of lymph node sections.
- It includes $n = 327,680$ image patches, categorized into $K = 2$ classes (metastatic tissue and normal tissue).

- Widely used in medical image analysis and for training models in detecting metastasis in tissue samples.

DOTA (Ding and Xia):

- A large-scale dataset designed for object detection in aerial images.
- It contains $n = 11,268$ images and 1,793,658 object instances, categorized into $K = 18$ object categories, including vehicles, buildings, and planes.

COCO [Common Objects in Context] (Microsoft):

- One of the most comprehensive datasets for object detection, segmentation, and image captioning tasks.
- It consists of $n = 330,000$ images (with over 200,000 labeled), 1.5 million object instances, $K = 80$ object categories, and 91 stuff categories.
- Additionally, each image includes 5 captions and keypoint annotations for 250,000 people, making it versatile for various computer vision tasks.

Data Labeling

Self-Annotating Domain-Specific Data:

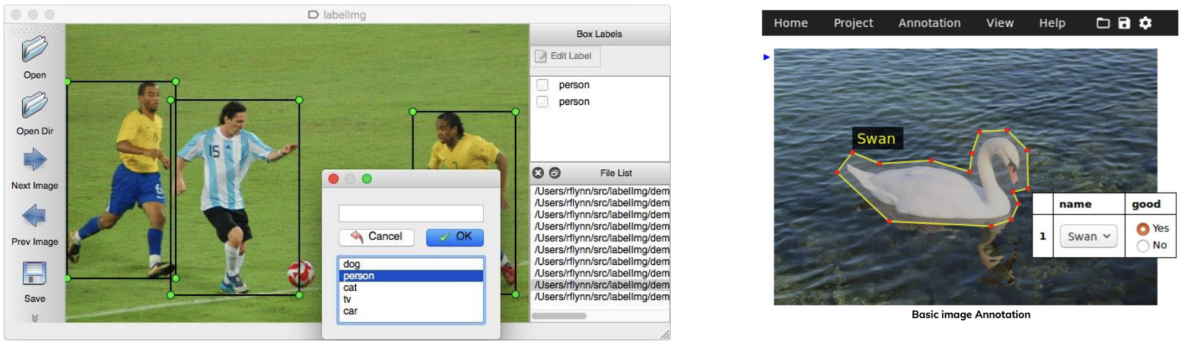


Figure 5.4: Enter Caption

- Numerous open-source and paid tools are available for image annotation, such as *LabelImg* and the *VGG Image Annotator*. These tools help create labeled datasets by generating bounding boxes or polygons around objects in images.
- **Assistance tools**, such as model predictions or automated bounding box suggestions, can be integrated to speed up the labeling process.

Considerations for Data Labeling

- **How much data is enough**: Depends heavily on the specific task. For example, detecting trucks in satellite images may require thousands of manually labeled examples.
- **Resource Limitations**: Projects are often constrained by available resources, such as budget, time, and human labor.
- **Who Labels the Data**:
  - *Project Team*: Researchers can label the data themselves.

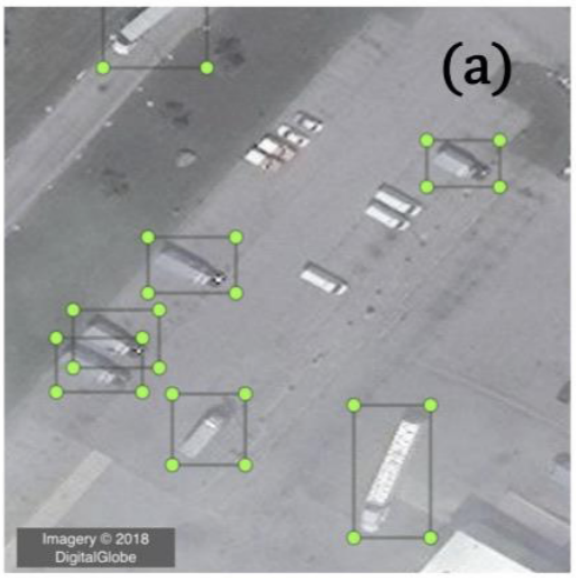


Figure 5.5: What is a truck, what is a van?

  - *Trained Research Assistants*: They can label the data more efficiently, especially in domain-specific contexts.
  - *Crowdsourcing Platforms*: Tools like *Amazon Mechanical Turk* enable large-scale labeling, but the quality of labeling can vary depending on the complexity of the task.

  Huge differences in quality depending on the task - some tasks can be translated to work with crowd sourcing (some can/should not)!

- **Quality Control**
  - **Annotation Scheme**: A well-defined, consistent annotation scheme is essential. Once defined, it should not be changed mid-process.
  - **Pilot Tests**: Conduct small-scale pilots to train annotators and test the annotation scheme.
  - **Edge Cases**: Resolve ambiguous examples (edge cases) with annotators to ensure consistent labeling.
  - **Inter-Annotator Agreement**: Monitor quality using inter-annotator agreement measures, such as Cohen's Kappa, to ensure label consistency.
  - **Publishing Data**: If allowed, publish the dataset to enable other researchers to use and extend the research.

Labeling with Active Learning

Active Learning:

- Active learning involves continuous annotation *while the model is being trained.*
- Learning algorithms query the user (often referred to as the "teacher") for labels when necessary, particularly in cases where the model is uncertain.
- The system focuses on edge cases, where the model struggles, to improve performance in areas of difficulty.

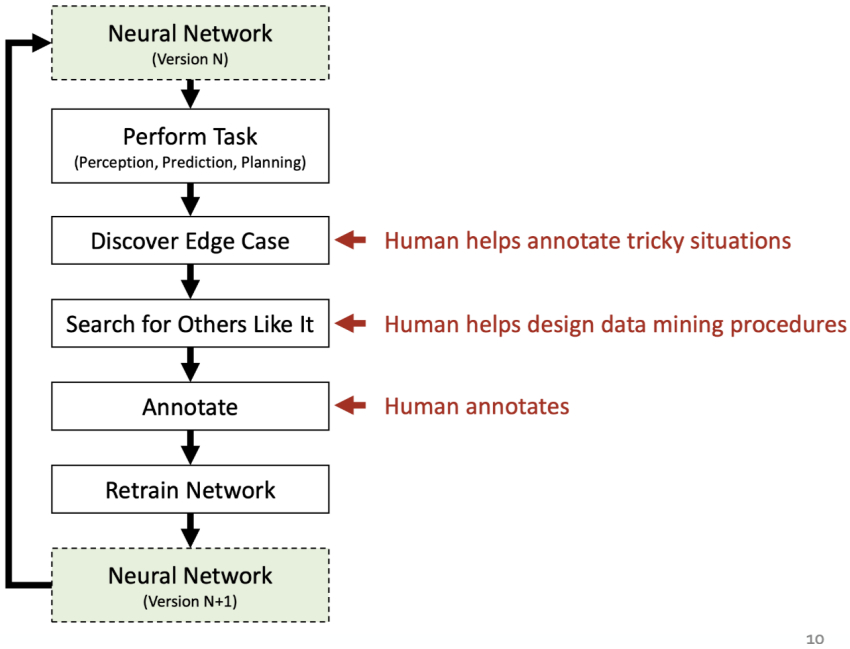


Figure 5.6: Process Flow

- **Pro:** this approach can reduce the overall amount of labeled data required for training.
- **Con:** there is a risk of oversampling uninformative examples.
  - If the initial training set lacks a sufficient number of typical examples and active learning is applied, the process may repeatedly focus on edge-case adjacent instances, pulling more of them from the unlabelled set.
  - This can distort the training set by overrepresenting less relevant or uninteresting examples.

> **On-line learning**:
>
> Similar to active learning: it has new data points coming in, but it's more passive as there's no 'human as teacher' component. It's just additional labeled data coming in.

Model-Assisted Labeling

Fast Segmentation:

- Models can be integrated into the annotation process to accelerate labeling tasks.
- In segmentation tasks, instead of manually drawing a polygon around an object, the annotator can simply click inside the object, and the model will suggest the appropriate boundaries.
- After the model's initial suggestion, manual correction may still be required to fine-tune the annotations.

5.1.1 Data Augmentation: generating existing examples

Data Augmentation

- Data augmentation involves generating additional training examples by applying **random transformations** to existing data. This enhances the model's **generalization capabilities** by introducing new variations without requiring manual data collection.

- It helps to **reduce overfitting** and improves performance on unseen data, **particularly in computer vision tasks**.

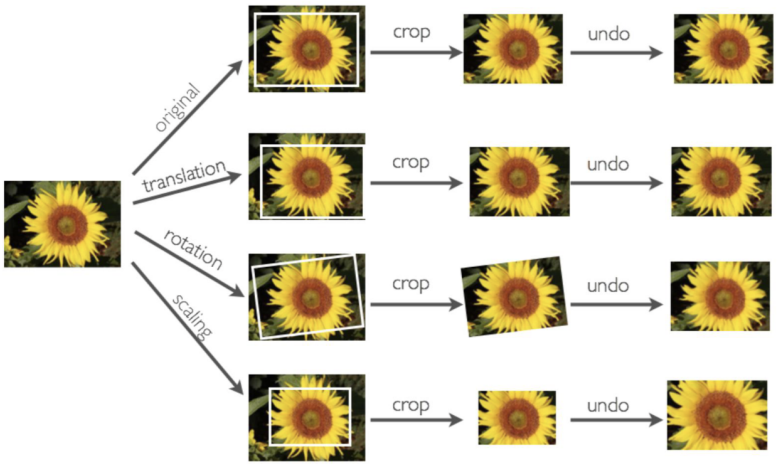


Figure 5.7: data augmentation

Generating Additional Examples

- **Concept**: Data augmentation can include operations such as *cropping*, *translation*, *rotation*, and *scaling*. These transformations create new images from existing ones, helping the model generalize better.

- These transformations are **only applied to the *training set*...**

- ...the **test set remains a valid evaluation** of performance.

- It is important to avoid applying data augmentation during the prediction phase to prevent introducing unnecessary variations.

Color Augmentation (for real-world applications)

- **Adjusting Brightness, Contrast, Saturation, and Hue**: By altering color properties, we can simulate different lighting conditions or environments in which objects appear.

- **Example**: In the case of a tiger image, augmenting brightness or contrast helps the model recognize tigers in various lighting conditions, making it more **robust to environmental variations**.

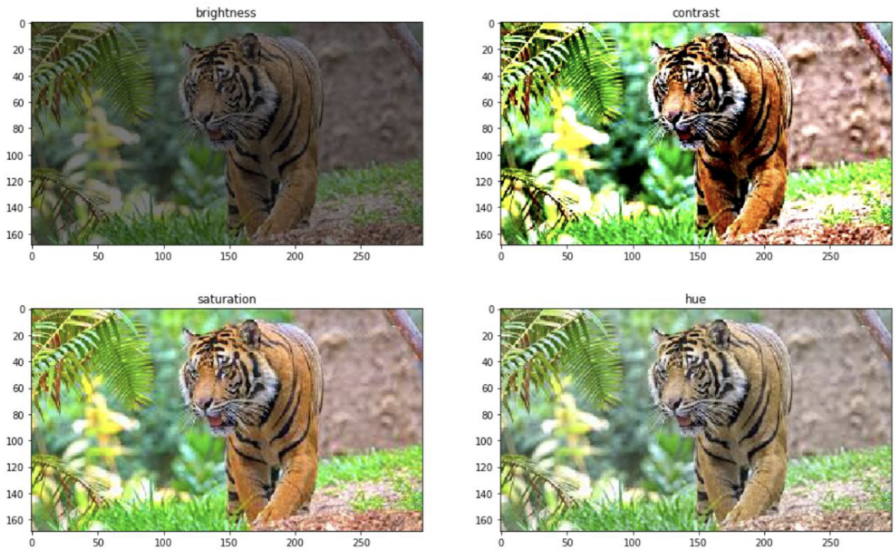


Figure 5.8: Adjusting Brightness, Contrast, Saturation, and Hue: This form of augmentation is particularly valuable in real-world applications where lighting is inconsistent or objects might appear under diverse conditions.

- **Example with Cat**: The slide shows color manipulations on a cat image across different color channels, **teaching the model to focus on structural details rather than relying on color information**.

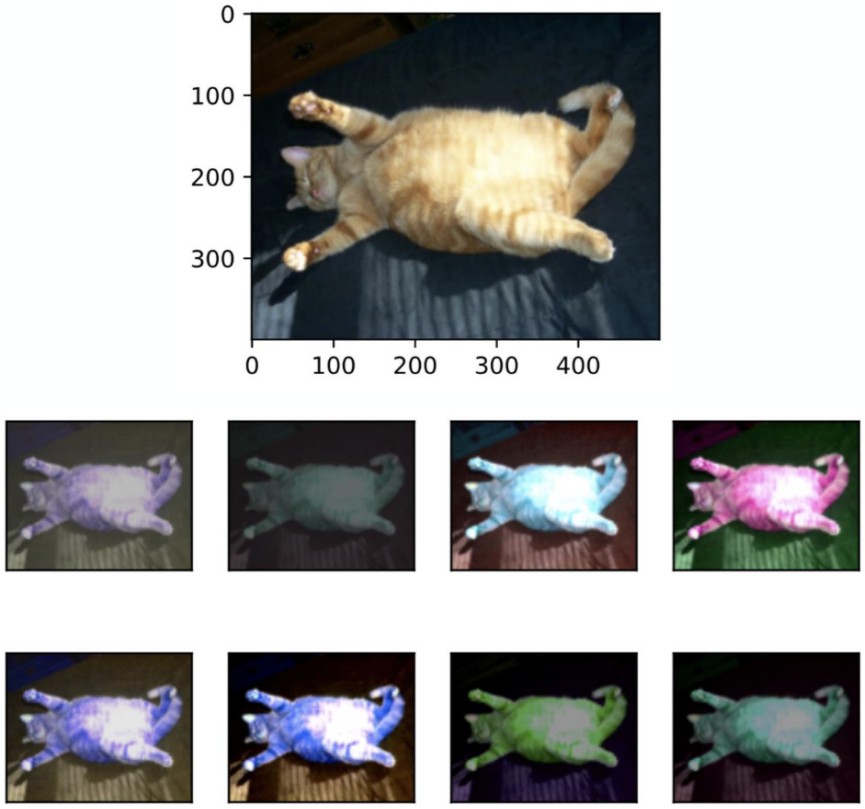


Figure 5.9: colour augmentation: teaches the model to focus on structural details rather than relying on color information

Elastic Distortions (for character recognition)

- **Purpose**: Elastic distortions are useful for tasks like character recognition (e.g., MNIST dataset). By introducing small elastic deformations, models become more robust to varied writing styles or imperfect representations.

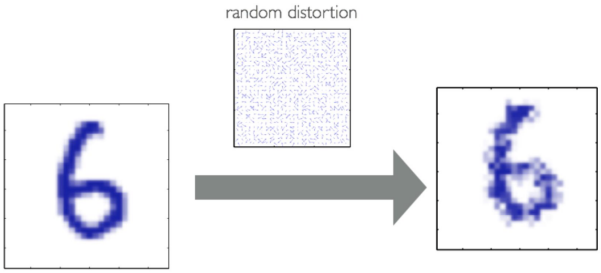


Figure 5.10: random distortion

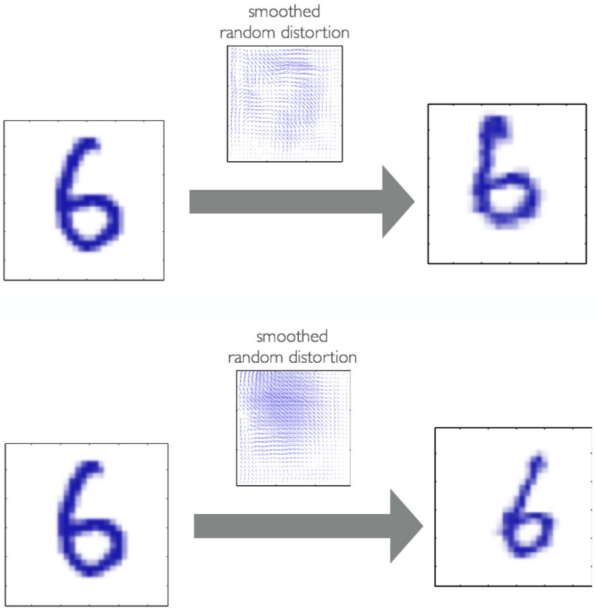


Figure 5.11: Smoothed random distortion

- **Mechanism**: A *distortion field* specifies how each pixel in the image is displaced, allowing for the simulation of slight alterations that mimic real-world variations.
- **Example**: The digit "6" is distorted to create a new training example, helping the model learn to recognize different variations of the digit.

Scaling

- Models do not natively handle scaling, which is particularly important to account for variations in scaling times.
- Note that augmentation is applied only to the test cases, not the training data.

Broader Context of Data Augmentation

- **Data Efficiency**: Augmentation is especially valuable when collecting new data is expensive (e.g., medical imaging, satellite data) or when only limited data is available.

- **Improved Robustness**: By introducing varied versions of the same object, models become more robust to real-world scenarios such as changes in lighting, orientation, or occlusion.

Why Data Augmentation is Powerful

- **Improves Generalization**: Models, particularly deep learning models like CNNs, tend to overfit small datasets. Data augmentation introduces controlled variation, reducing overfitting.

- **Cost-Effective**: Augmentation allows generating more data without additional data collection.

- **Versatility**: Modern deep learning frameworks provide various augmentation techniques (e.g., flipping, zooming, shearing), which can be applied together to simulate diverse conditions.

Conclusion:

- Convolution and pooling layers are responsible for *feature extraction*.
- The fully connected layers are the *prediction* component, learning patterns from the extracted features.
- Modern CNNs are now deep and narrow, consisting of many small filters stacked sequentially, whereas older architectures were wider and shallower.

5.2 Modern CNN models

VGG: Introducing Blocks and Deep Networks (2014)

Basic CNN Block:

- A basic CNN block consists of three components:

  1. A **convolutional layer** with padding to maintain the spatial resolution of the input image.
  2. A **non-linearity**, typically a *ReLU* activation function, to introduce non-linearity and help the model learn more complex patterns.
  3. A **pooling layer**, such as max-pooling, to downsample the feature maps and reduce the spatial dimensions.

- **Problem**: With many pooling layers, the resolution of the feature maps can reduce too quickly, causing loss of spatial information.

VGG Block:

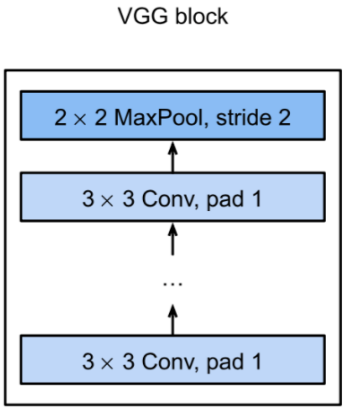


Figure 5.12: VGG Block

- VGG introduces the concept of *blocks* of layers rather than individual layers. Each block consists of:
  1. **Multiple convolutions** with small 3x3 kernels, keeping the height and width of the feature maps constant using padding.
  2. A **max-pooling layer** with a 2x2 kernel and stride 2 to halve the height and width after each block.
- **Shift in Thinking**: VGG popularized the idea of using **deep, narrow** networks with **small convolutions** (3x3) rather than shallow networks with large filters.
- **VGG-11 Example**: VGG-11 contains 8 convolutional layers and 3 fully connected layers at the end, with each block being followed by a pooling layer to gradually reduce spatial dimensions.

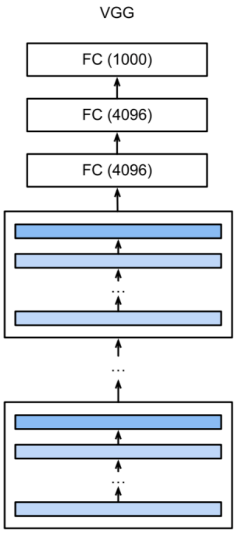


Figure 5.13: VGG

Impact of VGG:

- VGG's deep and narrow architecture allowed for much deeper networks that were computationally feasible, setting the foundation for future CNN architectures.
- VGG networks are often used as base models for transfer learning and fine-tuning on new datasets.

GoogLeNet: Inception Blocks (2014)

Combining idea of modular blocks alongside skip connections. So called *Inception block* from the movie, so-called as it allows you to keep going deeper.

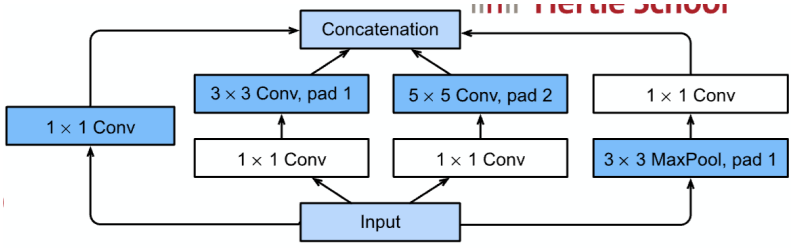


Figure 5.14: GoogLeNet: Inception Blocks

Inception Blocks:

- GoogLeNet introduced the *Inception block*, a multi-branch structure that allows the network to simultaneously process information at different scales:
  - The first three branches use 1x1, 3x3, and 5x5 convolutional filters to capture features at different spatial resolutions.
  - The fourth branch uses a 3x3 max-pooling operation to further capture context at a broader scale.
  - The output from all branches is concatenated along the channel dimension, producing a rich multi-scale representation.
- **1x1 Convolutions**: These are used primarily to reduce the number of channels before applying more computationally expensive 3x3 and 5x5 convolutions, helping to lower the overall computational cost while preserving important information.

1x1 Convolution Explanation:

- The only purpose of the 1x1 convolution is to change the number of channels (also known as feature maps) in a computationally efficient manner.
- This layer is followed by a non-linear activation function (e.g., ReLU), allowing the network to learn complex patterns without greatly increasing the model's size or computational demand.
- The 1×1 convolution layer adjusts the number of channels to match the outputs of the other branches and fine-tune the model's complexity. It maps the features to a single output map for each channel.
- Importantly, this isn't a traditional convolution because it doesn't exploit local spatial connectivity, but rather serves to adjust the number of channels and prepare the data for further parallel processing in subsequent layers. This operation introduces additional parameters to learn, often referred to as "weight parameters".

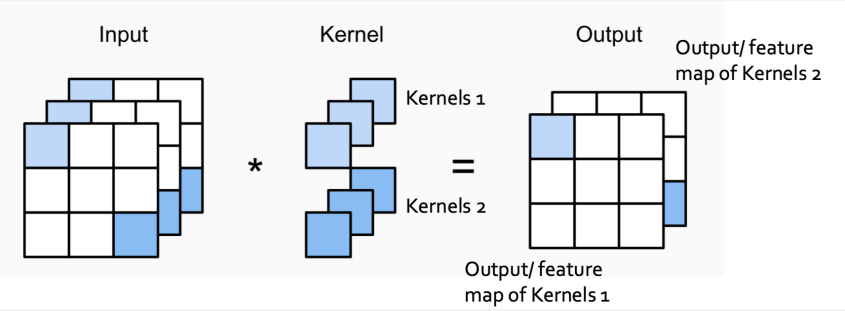


Figure 5.15:

If the number of input channels differs from the number of output channels, a single 1x1 convolution is used to adjust the number of channels:

- The 1x1 convolution will have a number of filters equal to the number of output channels required.
- So, if you need to match the input channels (let's say C_{in}) to the output channels (let's say C_{out}), the 1x1 convolution will have C_{out} filters (kernels), and this operation will transform the input to the desired output dimensions.

- After passing through the different branches, the outputs are concatenated along the channel dimension, forming a single tensor with multiple feature maps. Here, the concatenation refers to merging these parallel matrices into a higher-dimensional tensor, not just stacking vectors into a long one.

- The number of output channels in each branch is a hyperparameter. By increasing the number of output channels in a branch, we assign more weight to that branch in learning, which can influence the model's overall performance.

The most common operation is summing the results of the convolutions across all channels because this allows each filter to learn a weighted combination of the features from all input channels. The learned weights of the filter represent how much each channel contributes to detecting a particular feature.

For example, in an RGB image, the filter might learn to detect edges by combining the information from the Red, Green, and Blue channels in a particular way. The result from summing gives the final activation at that spatial location in the feature map.

Concatenation is generally used in specific architectures where different types of features are extracted from the input and need to be preserved separately.

A common scenario is in Inception modules in GoogLeNet or architectures that use multiple filter sizes or different processing operations in parallel (e.g., 1x1 convolutions, 3x3 convolutions, and max-pooling in parallel).

In these cases, different feature maps are created by different filters or operations, and then the feature maps are concatenated along the depth (channel) dimension to form a larger output volume.

For example, if a layer has 3 filters with 32, 64, and 128 channels, and each filter produces a feature map of size $H \times W$, the final output of the layer after concatenation will have a depth of $32 + 64 + 128 = 224$ channels.

1x1 Convolution: Technical Explanation

Role of 1x1 Convolutions:

- The 1x1 convolution is a key building block in modern CNN architectures like GoogLeNet.
- Its main purpose is to change the number of channels (feature maps) without affecting the spatial dimensions (height and width) of the input. This allows for reducing or expanding the depth of the network while keeping computational costs low.

How It Works

- A 1x1 convolution kernel takes each pixel in the input and multiplies it by a weight matrix, producing a new value for each output channel.
- Since the kernel size is 1x1, it does not affect the spatial dimensions (height and width) of the input feature map, but it does affect the depth (number of channels).
- After applying the convolution, the result is passed through a non-linearity, typically a ReLU activation function, to introduce non-linearity into the model.

Worked Example

Let's assume we have an input feature map of size $4 \times 4 \times 3$, where:

- The spatial dimensions are 4×4 (height and width).
- The depth (number of channels) is 3.

We want to use a 1x1 convolution to reduce the number of channels from 3 to 2.

- A 1x1 convolution kernel has the size 1×1, but since we want **to reduce the depth from 3 to 2, each kernel must have 3 weights (one for each input channel). Therefore, we have 2 kernels, each with a weight matrix of size** $1 \times 1 \times 3$**.**
- Mathematically, the output of the convolution is given by:

$$O_{i,j,k} = \sum_{c=1}^{3} (I_{i,j,c} \times W_{c,k})$$

  where I is the input feature map, W is the weight matrix (kernel), and O is the output feature map.

Step-by-Step Example:

1. **Input feature map** ($4 \times 4 \times 3$):

$$I = \begin{bmatrix} [1,2,3], & [4,5,6], & [7,8,9], & [10,11,12] \\ [13,14,15], & [16,17,18], & [19,20,21], & [22,23,24] \\ [25,26,27], & [28,29,30], & [31,32,33], & [34,35,36] \\ [37,38,39], & [40,41,42], & [43,44,45], & [46,47,48] \end{bmatrix}$$

   Each entry contains 3 channels.

2. **Convolution kernel** ($1 \times 1 \times 3$ for each output channel):

$$W_1 = [0.5, 0.3, 0.2], \quad W_2 = [0.1, 0.2, 0.7]$$

   These kernels will reduce the number of channels from 3 to 2.

3. **Apply the convolution for each output channel**:

Multi-Branch Networks: Impact of Inception Block:

- The Inception block's structure allows for more complex feature extraction **across different spatial scales**, improving the model's ability to **recognize objects regardless of their size or position**.
- All four branches use padding to ensure that the height and width of the input and output remain constant, making it easier to concatenate the outputs.
- **Hyperparameter Tuning**: One of the key design choices in Inception blocks is deciding how many output channels to allocate to each branch. This determines the model's capacity to handle different-sized features.
- **Efficiency and Accuracy**: Despite its complexity, GoogLeNet was computationally more efficient than earlier models while still achieving state-of-the-art performance, especially in image classification tasks (e.g., ImageNet).

Conclusion: VGG vs. GoogLeNet:

- **VGG** represents a shift towards **deeper and narrower** networks with small convolutions,
- while **GoogLeNet** focuses on **multi-scale** feature extraction with Inception blocks, balancing computational efficiency with model accuracy.

Residual Networks (ResNet) - Adding Skip Connections (2016)

Took the idea of the GoogLeNet, simplified it and added some theory.

Idea: We want a larger network to be at least as good as the smaller one.

Further exploiting the idea of **skip-connections**: adding the identity function to the new layer will make it at least as effective as the original model.

- At any point we have the same information as in the previous layers, so that the new layers can only make it better - we are not throwing away info.
- Can overcome vanishing gradients (by providing a direct path for gradients to flow backward during training) to build deeper models.

Introducing the **residual block!** (Can think of it as a 2-branch version of the inception block) - 1 branch keeps things as they are; 1 branch tries to learn and improve.

The network learns the *residual*, or the difference between the desired output and the input, which is generally easier to learn than mapping directly from input to output.

Motivation for Residual Connections:

- Deep networks tend to suffer from the ***vanishing gradient problem***, where the gradients become too small to effectively train deeper layers.
- The core idea of *residual connections* is to allow layers to learn modifications to the identity function, ensuring that the network performance does not degrade as it grows deeper.

- By **adding the identity function through a skip connection**, the network ensures that the deeper layers can at least perform as well as the shallower network.
- The key point here is that instead of concatenating feature maps, as done in networks like GoogLeNet, ResNet simply adds the input to the output. For this to work, the number of channels in the input and output must match.
  - Why? because the residual connection involves an element-wise addition of the input and output:

    $$\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{x}$$

    By learning residuals (differences) instead of full transformations, the network avoids the vanishing gradient problem, which helps with training deeper networks. The idea of residual learning is to allow the network to learn identity mappings more easily, which can be viewed as learning "what should change" in the input, making deeper networks more efficient and easier to train.
  - The 1×1 convolutional layer is used to match the number of channels between the input and output of the residual block.
  - If the input has a different number of channels compared to the output, the 1×1 convolution adjusts the number of channels in the input (by applying a linear transformation) so that it matches the output size before adding them together.
  - The 1x1 convolution will have a **number of filters equal to the number of output channels** required.
  - This operation ensures that the addition is dimensionally valid.

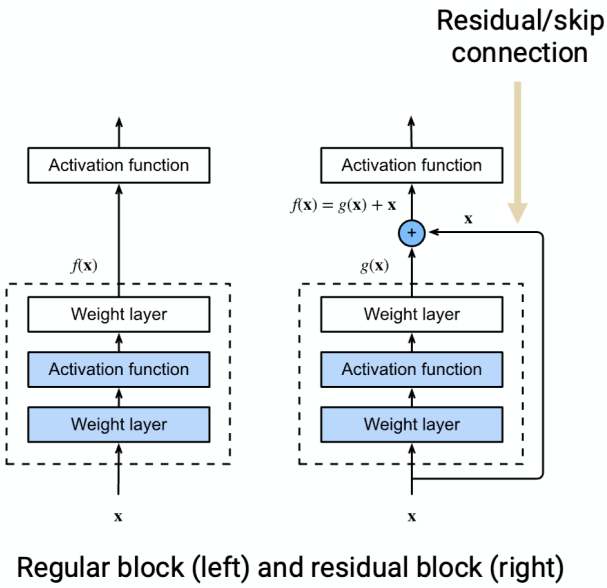


Figure 5.16: the residual block (portion of the block in dotted lines) learns the **residual mapping** $g(x) = f(x) - x$.

Residual Block Structure:

- A residual block consists of a weight layer followed by an activation function. The residual connection skips this operation and adds the input x directly to the output *after applying the non-linear function* $f(x)$.

- The residual mapping is expressed as:

$$f(x) = g(x) + x$$

  where $g(x)$ represents the output of the block after passing through the convolutional and activation layers, and x is the input.

- This architecture allows the network to focus on **learning the *residual*, or the difference between the desired output and the input**, which is generally easier to learn than mapping directly from input to output.
  - if $f(x)$ is close to x, the residual is much easier to learn.
- Can be thought of as a special 2-branch case of the inception block.

Key Benefits of Residual Connections:

- **Easier to optimize**: Since the residuals are typically small and easier to learn, residual networks can be optimized more effectively, especially as the network depth increases.
- **Prevents performance degradation**: Adding residual connections ensures that deeper networks do not perform worse than shallower ones, addressing the degradation problem in deep learning.
- **Modular and scalable**: ResNet architectures can be scaled to hundreds or even thousands of layers, such as ResNet-152, without encountering optimization difficulties.

ResNet Block Variants

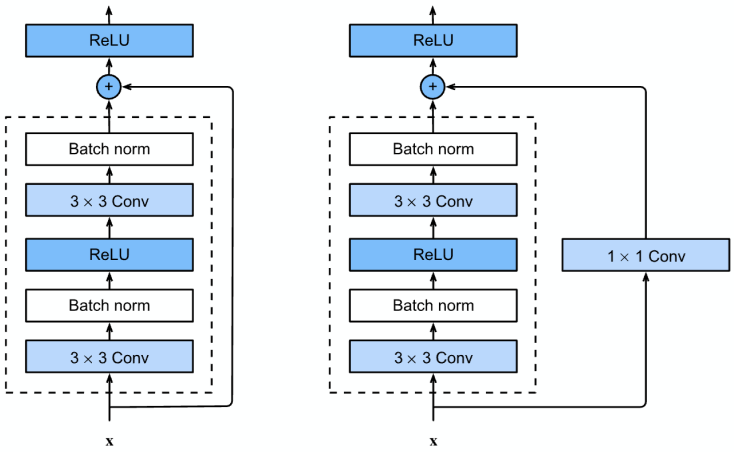


Figure 5.17: Standard ResNet Block

Standard ResNet Block:

- A standard ResNet block consists of two 3x3 convolutional layers, followed by batch normalization and ReLU activation.
  - Inspired by VGG block (a sequence of convolutions with 3×3 kernels, followed by max-pooling; emphaises deep, narrow networks)
- The input x is passed through these layers and added back to the block's output, ensuring that both the learned transformations and the input contribute to the final output.

Bottleneck ResNet Block (1x1 Convolutions):

- In deeper networks (e.g., ResNet-50, ResNet-152), a bottleneck design is used to reduce the computational cost.

- The bottleneck block includes a 1x1 convolution at the start and end of the block to reduce and then restore the number of channels, effectively lowering the computational load.

- This block is particularly useful when the 3x3 convolutions in the residual block need to change the number of channels, as the 1x1 convolution in the residual connection ensures that the dimensions match.

Bottleneck ResNet Block (1x1 Convolutions)
Purpose of the Bottleneck Block:

- In very deep networks, such as *ResNet-50* or *ResNet-152*, the computational cost of performing multiple 3x3 convolutions can become very high, especially when the number of channels is large.
- The bottleneck block introduces two 1x1 convolutions to reduce this cost:
  - A 1x1 convolution at the start reduces the number of channels, making the subsequent 3x3 convolution more efficient.
  - A second 1x1 convolution at the end restores the number of channels back to the original, ensuring that the output matches the input dimensions.

Worked Example

Let's assume we have an input feature map of size $56 \times 56 \times 256$ (height, width, and depth). We want to perform a residual operation using a bottleneck block with the following structure:

- 1x1 convolution to reduce the channels from 256 to 64.
- 3x3 convolution to operate on these reduced channels.
- 1x1 convolution to expand the channels back to 256.

Why the Bottleneck Block is Efficient

- Without the bottleneck structure, performing a 3x3 convolution on a 56x56x256 input would require significantly more computations.
- By first reducing the number of channels with the 1x1 convolution, the cost of the 3x3 convolution is greatly reduced.
- Restoring the channels after the 3x3 convolution ensures that the network can still learn from a high-dimensional feature space, without compromising on the ability to add residual connections.

Conclusion:

- The bottleneck block enables ResNet models to scale to greater depths (e.g., ResNet-50, ResNet-101, ResNet-152) by significantly reducing computational costs.
- The 1x1 convolutions act as efficient transformations that reduce and restore the number of channels, allowing the network to process feature maps at different scales while maintaining overall efficiency.

Bottleneck ResNet Block (1x1 Convolutions)
Step-by-Step Process:

1. **Input feature map**:

   $$\text{Input size: } 56 \times 56 \times 256$$

   Each pixel in the 56x56 spatial grid has 256 channels.

2. **1x1 Convolution (Channel Reduction)**:

   - The first 1x1 convolution reduces the depth of the input from 256 channels to 64 channels.
   - The spatial dimensions (56x56) remain unchanged, but the number of feature maps (channels) is reduced:

     $$\text{Output size: } 56 \times 56 \times 64$$

   - This reduction lowers the computational cost for the next 3x3 convolution, as the number of channels to process has been reduced by a factor of 4.

3. **3x3 Convolution**:

   - Next, we apply a 3x3 convolution with 64 input channels, which is computationally less expensive compared to operating on 256 channels.
   - This convolution will preserve the spatial dimensions but may extract more complex features:

     $$\text{Output size: } 56 \times 56 \times 64$$

4. **1x1 Convolution (Channel Restoration)**:

   - The final 1x1 convolution expands the number of channels back to the original 256 channels to ensure that the output matches the input dimensions:

     $$\text{Output size: } 56 \times 56 \times 256$$

   - This step ensures that the residual connection can be added back to the original input, which had 256 channels.

5. **Residual Connection**:

   - The input feature map (56x56x256) is directly added to the output of the bottleneck block (56x56x256), resulting in the final output:

     $$\text{Final output size: } 56 \times 56 \times 256$$

   - This residual connection ensures that any skipped information from the original input is retained, allowing the network to effectively "reuse" the input data if needed.

ResNet-18 Example

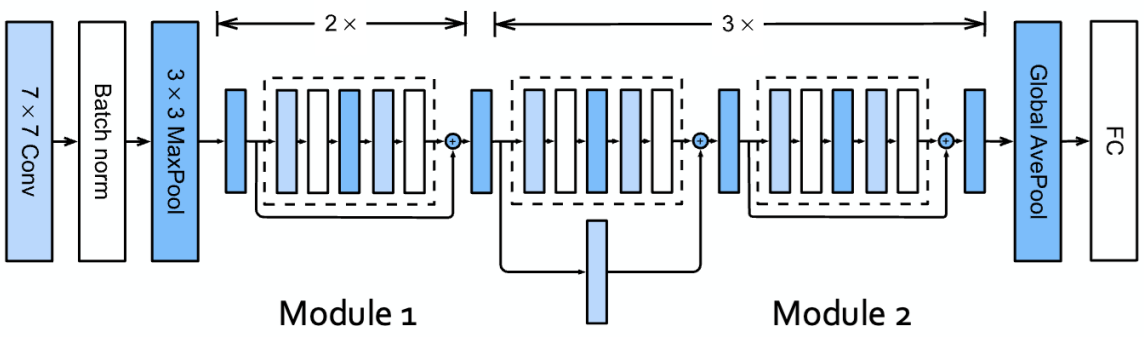


Figure 5.18: ResNet18

Network Structure:

- **ResNet-18** consists of:
  - An initial 7x7 convolutional layer followed by a 3x3 max-pooling layer.
  - 4 modules, each consisting of a series of residual blocks.
    - 2 × Module 1 (1 block)
    - 3 × Module 2 (2 blocks)
    - Each residual block has two convolutional layers.
  - A final fully-connected layer
- In ResNet-18, Module 1 consists of 2 blocks, while Module 2 consists of 3 blocks. The total number of layers with learnable weights can be calculated as:

$$2 \times 2 + 3 \times (2 + 2) + 1 + 1 = 18$$

Global Average Pooling:

- After the convolutional layers, ResNet uses *global average pooling*, which reduces each feature map (channel) to a single number by averaging the spatial dimensions.
- This method is *more efficient than fully connected layers, reducing the likelihood of overfitting.*
- The output of the global average pooling is fed directly into the softmax layer for classification.
- Need K feature maps for K classes. (In essence each feature map correspeonds to a discriminator for that class)

Global Average Pooling Explanation

A type of pooling operation used at the **end of convolutional neural networks**, typically before the final classification layer (softmax).

*It works by averaging each feature map (or channel) over its **entire spatial dimensions** (height and width), reducing each feature map to a single value.*

Why Use Global Average Pooling?

- **Prevents Overfitting**: Unlike fully connected layers, which have a large number of parameters and can lead to overfitting, global average pooling has no learnable parameters. This makes it more robust and less prone to overfitting, especially when the training data is limited.
- **Efficiency**: GAP reduces the feature maps to a small fixed-size output (1 value per channel), making it computationally efficient and reducing the number of parameters.
- **Direct Connection to Classes**: In the case of classification, each of the C feature maps (channels) can be thought of as a representation for one of the K classes, with the pooled value representing the overall activation for that class.
- **Classification**: The output of the global average pooling layer, which is $1 \times 1 \times C$, is directly fed into a *softmax* layer for classification. Here, $C = K$, where K is the number of classes. Each pooled value is used to predict the probability of the corresponding class.

Efficiency Compared to Fully Connected Layers

- In traditional CNNs, fully connected (FC) layers are often used after the convolutional layers. However, fully connected layers introduce a large number of parameters, especially when the input feature maps are large, which increases the risk of overfitting.
- For example, if we used an FC layer on a $7 \times 7 \times 512$ feature map, it would involve connecting all 25,088 values to the next layer, leading to a large number of learnable parameters.
- In contrast, global average pooling significantly reduces this to just 512 values (one for each channel), which are passed directly to the softmax layer, making the network both more efficient and less prone to overfitting.

Conclusion

- Global average pooling simplifies the transition from feature extraction to classification by reducing each feature map to a single value, while preserving the global information from the entire spatial dimension.
- It is particularly useful in reducing the model size and avoiding overfitting, making it a popular choice in deep CNN architectures like ResNet.

Global Average Pooling: How It Works

- Assume the feature map from the last convolutional layer has dimensions of $H \times W \times C$, where H is the height, W is the width, and C is the number of channels (feature maps).
- Global average pooling computes the mean of each $H \times W$ feature map, reducing the spatial dimensions to 1x1. Thus, the output will have dimensions $1 \times 1 \times C$.
- Mathematically, for each channel c, the global average pooling operation is given by:

$$\text{GAP}(c) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} x_{ijc}$$

  where x_{ijc} is the value at position (i, j) in the c-th feature map.

Worked Example

- Suppose we have a final feature map of size $7 \times 7 \times 512$ coming from the last convolutional layer of a CNN (e.g., ResNet). This means:
  - There are 512 channels.
  - Each channel (feature map) is of size 7×7.
- For each channel, global average pooling computes the average of all the 7×7 values. The result for each channel is a single value (1x1), resulting in an output of size $1 \times 1 \times 512$.
- This reduces the entire feature map to just 512 values, one for each channel, irrespective of the spatial dimensions.
- The key point is that no spatial information is retained, but instead, global information from the entire feature map is summarized by the average.

Summary of ResNet

- ResNet allows for the training of very deep networks (e.g., ResNet-152) by introducing residual connections, making it possible to avoid the vanishing gradient problem.
- These connections ensure that layers can be added without causing degradation, as the identity mapping is always preserved, ensuring performance doesn't degrade with increased depth.
- Very common as pre-trained models for fine tuning: ResNet models are widely used in transfer learning due to their scalability and effectiveness in learning complex representations.

Fine Tuning Pretrained Models

Motivation for Fine-Tuning:

- **Resource Efficiency**: Training large neural networks from scratch requires significant computational resources, including time, energy, and data. Fine-tuning offers a cost-effective alternative by reusing models that have already been trained on large datasets.

- **Basic Image Features Can Be Quite Generic**: Large models, like those trained on ImageNet, learn general-purpose features in their early layers, such as edges, textures, and shapes, which can be transferred to other tasks or domains.
- **Limited Target Data**: In many cases, the target domain (task) may have limited labeled data, making training from scratch impractical. Fine-tuning allows us to leverage knowledge learned from a large source dataset (like ImageNet) and adapt it to the target domain.
- **Transfer Learning**: Fine-tuning is a type of transfer learning, where the knowledge learned from a source dataset is transferred to a different, but related, target dataset. For example, a model trained on ImageNet for object recognition can be fine-tuned to recognize lesions in medical images.

Example: Digital Pathology

- Even though the source dataset (ImageNet) contains generic objects such as animals and vehicles, the general visual features learned by the model can still be useful when applied to medical imaging tasks (target dataset).
- ImageNet has v little info in it about rat tissue, but the edges and other feature extraction parts it learnt are v useful.
- For instance, a pre-trained model may still perform well on identifying specific patterns, such as lesions in tissue samples, after fine-tuning on a smaller medical image dataset.

General Procedure for Fine-Tuning

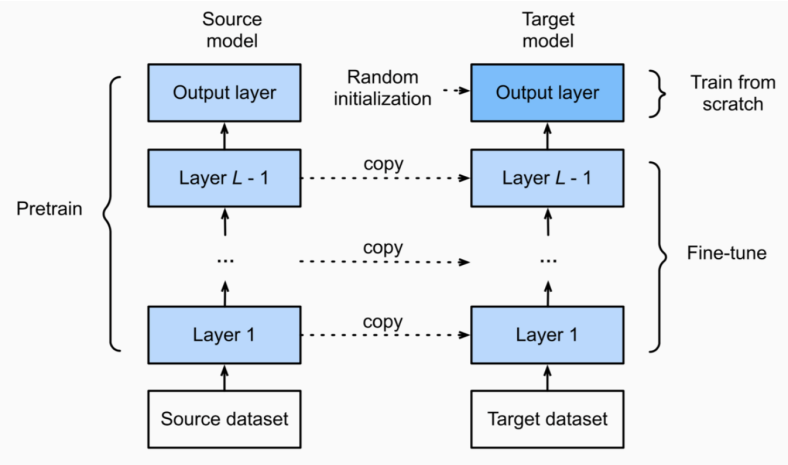


Figure 5.19: Fine-tuning

1. **Pretrain on Source Dataset**: Train a neural network model on a large dataset (e.g., ImageNet) or download a pre-trained model that has been optimized on such a dataset.
2. **Create Target Model**:
   - Copy all the layers and parameters from the source model to the target model, *except for the final output layer*.

- This ensures that the *feature extraction layers of the model are retained*, which capture general image features.

3. **Add New Output Layer**:

   - Replace the output layer of the pre-trained model with a new output layer that matches the number of categories in the target dataset.
   - The parameters of this new output layer are initialized randomly, as it has not been trained on the target dataset.

4. **Train on Target Dataset**:

   - Fine-tune the model by training on the target dataset. During this process:
     - The new output layer is trained from scratch.
     - The parameters of the earlier layers are fine-tuned, using the knowledge from the source model as a starting point.
   - Often, the lower layers (closer to the input) are "frozen" (i.e., their parameters are not updated), as they contain very general features that do not need to be retrained.

Why Fine-Tuning is Effective

- **Reusing Pre-trained Knowledge**: Instead of training a model from scratch, we use a model that has already learned useful representations, such as edges or shapes, from a large dataset. This knowledge is then fine-tuned to better match the specific patterns in the target dataset.

- **Avoids Overfitting**: Because only a small portion of the model (e.g., the output layer) is trained from scratch, fine-tuning helps prevent overfitting to small target datasets.

- **Speeds Up Training**: Since only the output layer and upper layers are retrained, the model converges faster than it would if it were trained from scratch.

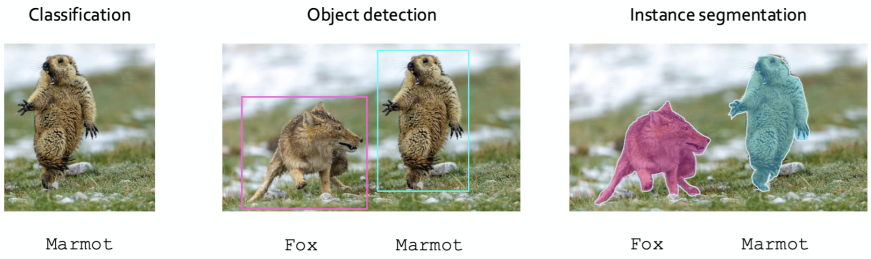
Fine-Tuning in Practice

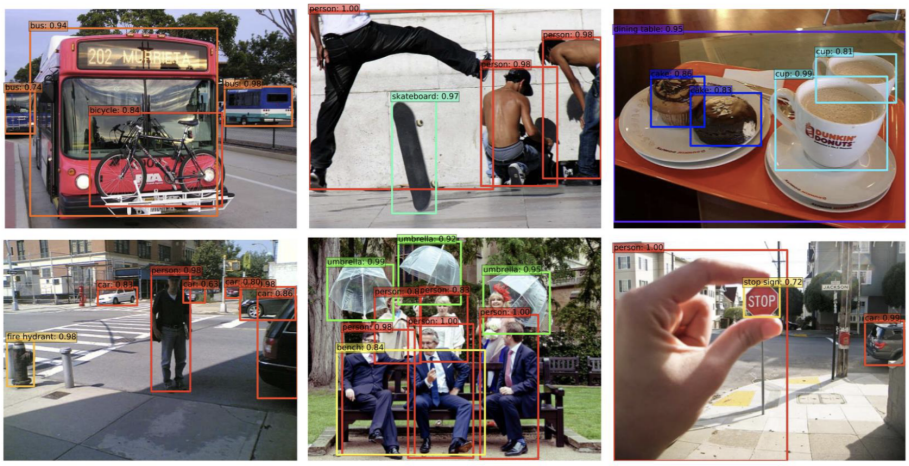
- **Common in Transfer Learning**: Fine-tuning is widely used in transfer learning tasks, especially in domains where labeled data is scarce. This includes applications in medical imaging, natural language processing, and many specialized vision tasks.

- **Model Freezing**: In many cases, the lower layers of the model are "frozen" to reduce the computational cost of fine-tuning. This prevents updating weights in layers that already capture general features, focusing training on the higher, task-specific layers.

- **Pretrained Models for Fine-Tuning**: Many pre-trained models, such as ResNet or VGG, are available in deep learning libraries (e.g., TensorFlow, PyTorch) and can be easily downloaded and fine-tuned for various tasks.

5.3 Object Detection





Overview

Object detection (or 'object recognition') refers to the process of identifying objects within an image or video and determining their **classes**, **positions**, and **boundaries**. It involves not just classifying an object but also *locating it* in the image through **bounding boxes**.

Tasks in Object Detection

- **Find objects in the image:** The model must identify the relevant objects present in the image. In some cases, this could involve multiple objects, each of a different class.

- **Draw (tight) bounding boxes:** A bounding box is a rectangle drawn around the object of interest to mark its spatial location. The model needs to determine the coordinates of the box in relation to the entire image.

- **Determine the class of the object:** The model classifies the object into one of the pre-defined categories based on the features it has learned during training.

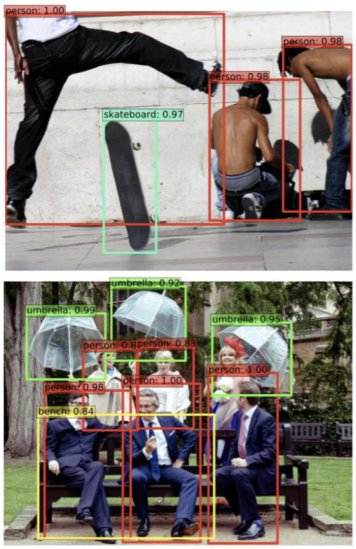


Figure 5.20: multiple objects

Applications of Object Detection

- **Remote sensing:** Identifying objects (e.g., planes, cars) from satellite images.
- **Autonomous vehicles:** Detecting pedestrians, cars, and other obstacles.
- **Face recognition:** Identifying people in crowds for surveillance or social media tagging.
- **Traffic monitoring:** Counting vehicles for congestion analysis.

Bounding Boxes Representation

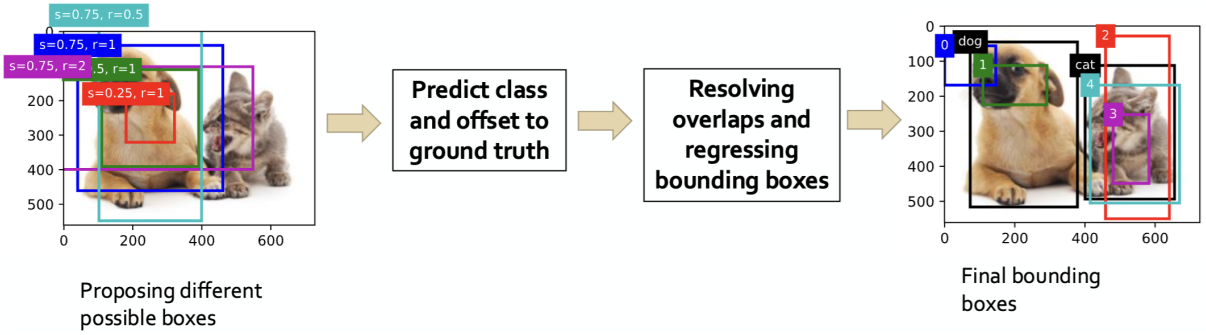
Bounding boxes are typically represented using two common methods:

- **Corner-based:** This representation uses the coordinates of the upper-left (x_1, y_1) and lower-right corners (x_2, y_2) of the box.
- **Center-based:** Alternatively, we can represent the bounding box using the center coordinates (x, y) of the box along with its width and height (w, h).

Boxes are learned with training data with ground truth bounding boxes.

Basic Workflow

- First, the model proposes several possible boxes across the image at different locations and scales (proposing anchor boxes).
- Then, the model predicts the class of the object in each box and adjusts the box to fit the object more tightly.
- Finally, overlapping boxes are resolved by a method called non-maximum suppression (NMS), which keeps only the most confident box.



Anchor Boxes:

An anchor box is a pre-defined bounding box with a fixed size and aspect ratio that is applied to different parts of the image to detect objects. These anchor boxes help the model handle objects of different sizes and aspect ratios effectively.

Anchor boxes act like a grid: they provide possible regions where an object might be located, regardless of whether an object is there or not. For each box, the model evaluates whether it contains an object, which class the object belongs to, and how the box should be adjusted to tightly fit the object (using predicted offsets).

The purpose of using anchor boxes is to provide the model with pre-defined regions to check for objects, instead of having it scan the entire image pixel-by-pixel. This makes the detection process computationally efficient, as the model can focus on adjusting these predefined boxes instead of trying to create new boxes from scratch for every possible object.

1. Predicting Boxes

Anchor Boxes

The model starts by suggesting several potential bounding boxes (referred to as anchor boxes) at various locations and scales over the image. These boxes serve as starting points for detecting objects.

- Object detectors start by proposing a number of **anchor boxes** at different scales $s_1, \cdots, s_n$ and aspect ratios $r_1, \cdots r_m$.
- For computational efficiency, a small set of these anchor boxes is chosen, and each of them is evaluated at different points in the image.
- For each anchor box, the model predicts both the class of the object and an offset to the anchor box, adjusting the box to more accurately fit the object.
- A predicted bounding box is thus obtained according to an anchor box with its predicted offset

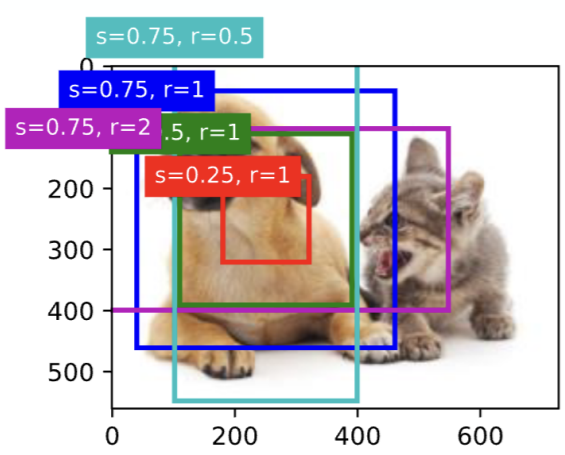


Figure 5.21: predicted bounding boxes

2. Class Prediction (based on a predicted box)

For each proposed anchor box, the model predicts: (1) The class of the object (if any) that might be present within the box.(2) Adjustments to the box (called offsets) to make it more tightly fit the detected object.

- The model assigns a confidence score to each box, representing how likely the object in the box belongs to a particular class.
- The highest confidence score is typically used to select the final predicted class for each box.

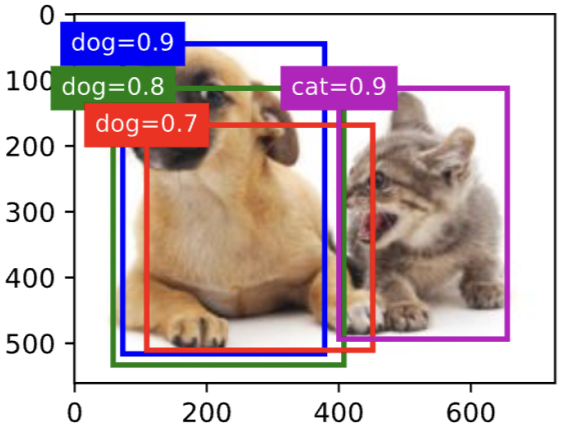


Figure 5.22: class prediction

Once the anchor boxes are adjusted, the model predicts the class of the object in each box. This prediction comes with a **confidence score** — a measure of how likely the object belongs to a certain class. Confidence scores are used to match predicted box to ground truth. For example, a score of 0.9 might indicate a 90% confidence that the object is a car. The box with the highest confidence score is usually selected as the final bounding box for that object.

3. Resolving Overlaps: Non-Maximum Suppression (NMS)

After predictions are made, many boxes may overlap. The model uses a process called non-maximum suppression (NMS) to keep only the box with the highest confidence score, discarding the rest.

- NMS helps to remove duplicate bounding boxes that refer to the same object.
- The algorithm selects the box with the highest confidence and removes all other boxes that overlap with it beyond a certain IoU threshold.
- This process is repeated until all overlapping boxes are removed, ensuring that only one box per object remains.

Measuring Agreement: Intersection over Union (IoU)

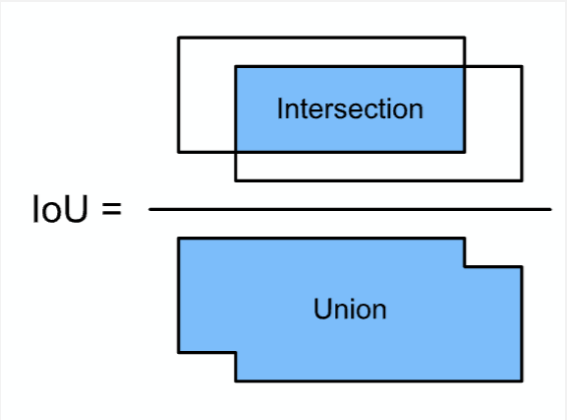
Object detectors learn by comparison of predicted bounding boxes with ground truth bounding boxes

How do we measure agreement? **Jaccard Index**

$$J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}$$

We can measure the similarity of the two bounding boxes by the Jaccard index of their pixel set (Intersection over Union (IoU))

- IoU measures the overlap between the predicted bounding box and the ground truth bounding box.
- It is calculated as the area of the intersection between the two boxes divided by the area of their union.
- IoU helps to quantify how well the predicted box matches the ground truth box. A higher IoU means a better match.



1. Select the predicted bounding box with the **highest confidence**.
   - **Remove all other predicted bounding boxes whose IoU with that first box exceeds** a predefined threshold (itself a hyperparameter).
2. Select the predicted bounding box with the second highest confidence.

- **Remove all other predicted bounding boxes whose IoU with that first box exceeds** a predefined threshold.

3. Repeat until all the predicted bounding boxes have been used.

Now, the IoU of any pair of predicted bounding boxes is below the threshold; no pair is too similar with each other.

SSD: Single Shot MultiBox Detector

SSD by Liu et al (2015)

- A computationally efficient (fast) and high-performing object detector for multiple categories.
- As accurate as slower techniques that perform explicit region proposals and pooling (including Faster R-CNN).
- Models relying on region proposals have an entire network to propose suitable bounding boxes for further classification, making them slow.
- SSD uses *anchor boxes* to propose a number of boxes at low computational cost.

Non-Maximum Suppression (NMS)

Non-Maximum Suppression is a technique used to eliminate redundant bounding boxes that overlap significantly and represent the same object.

In object detection, multiple bounding boxes may predict the same object with slight variations. Here's how NMS helps resolve these overlaps:

- **Problem**: Many predicted bounding boxes with similar positions and dimensions can overlap significantly, leading to multiple detections for the same object.
- **Solution (Non-Maximum Suppression)**:
  1. Select the bounding box with the highest confidence score.
  2. Remove all other bounding boxes that have a high Intersection over Union (IoU) overlap with the selected box, based on a predefined threshold.
  3. Repeat this process until all bounding boxes are processed.
- **Effect**: This leaves only the bounding boxes with the highest confidence for each detected object, ensuring no redundant boxes are retained.

SSD: Single Shot MultiBox Detector (2015)

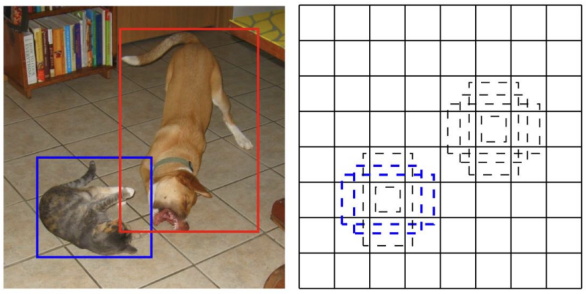
SSD by Liu et al. is a fast and efficient object detector that allows real-time detection *without region proposal stages*.

- **Efficiency**: SSD is computationally efficient and performs detection in a *single pass*, unlike methods that require multiple passes over the image.
- **Accuracy**: Comparable to slower methods like Faster R-CNN, SSD achieves similar accuracy by directly predicting bounding boxes and classes at different feature map scales.
- **Anchor Boxes**: SSD uses a fixed set of anchor boxes at various scales, which helps in detecting objects of different sizes.

Anchor Boxes

Anchor boxes are predefined bounding boxes with various sizes and aspect ratios, centered at grid points across the feature map. Here's how SSD uses them:

- **Grid Representation**: Imagine the input image divided into a grid. Each grid cell can predict multiple bounding boxes based on the anchor boxes.
- **Predictions**: For each anchor box, SSD predicts offsets (to adjust position and size) and class scores for different object categories.



(a) Image with GT boxes (b) 8×8 feature map

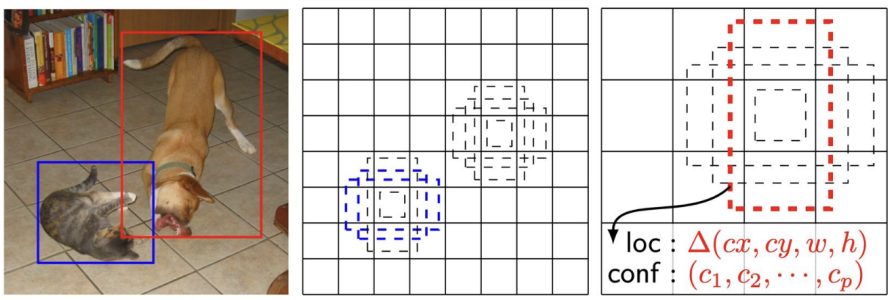
Multiscale Anchor Boxes

SSD employs different feature maps at different scales to capture objects of varying sizes effectively:

It uses the *same set of anchor boxes* but on a *different (lower resolution) feature map* higher up in the model (where there has been some pooling happening. So now the same kind of anchor box set gives rise to larger input image areas.

All of the different scaled anchor boxes get thrown into the same pool and then subjected to non-maximum suppression.

- **Different Scales**: SSD uses feature maps of *decreasing resolution* (e.g., 8×8, 4×4, etc.) to detect larger objects in lower-resolution maps and smaller objects in higher-resolution maps.



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

- **Uniformly Distributed Anchor Boxes**: Each feature map level has anchor boxes distributed uniformly, ensuring coverage across scales.

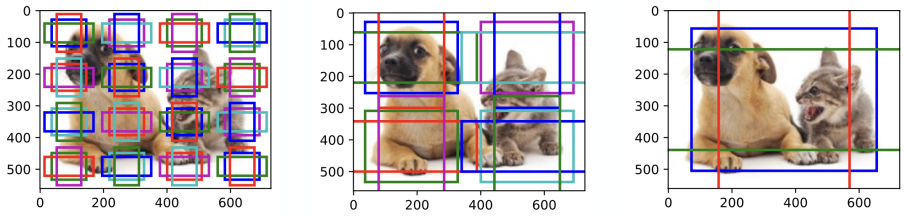


Figure 5.23: WWe have different feature maps at different scales; Anchor boxes are uniformly distributed over each feature map; This allows to create bounding boxes at different scales

Prediction in SSD

SSD makes predictions in **one pass** by outputting **two main pieces of information** for each anchor box:

1. **Offset Prediction**: The network predicts offsets to adjust each anchor box's size and position to better fit the object.
2. **Class Confidence Scores**: For each anchor box, SSD provides confidence scores for every object class, enabling classification.

SSD Architecture

The SSD architecture builds on a base network (such as truncated VGG-16) and includes additional convolutional layers to progressively decrease the spatial resolution while increasing the depth of feature maps. This allows for multiscale predictions:

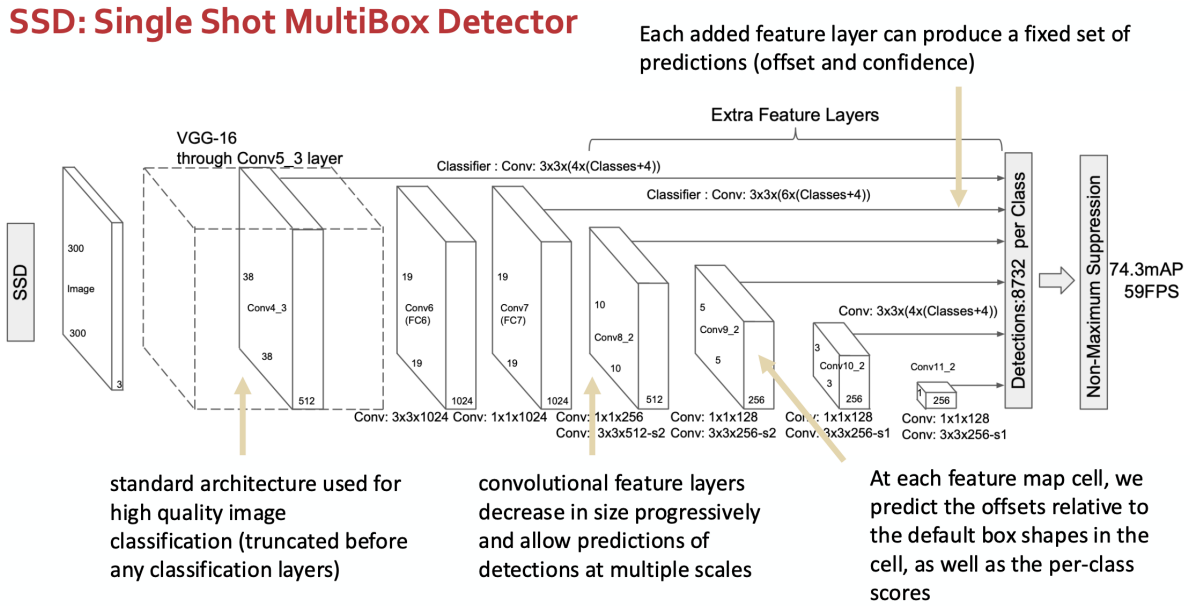


Figure 5.24: SSD Architecture

- **Base Network**: Uses a standard image classification network (e.g., VGG-16) up to a certain layer as a feature extractor.
- **Additional Convolutional Layers**: These layers reduce the feature map size and increase depth, enabling the network to detect objects at various scales.
- **Fixed Set of Predictions**: Each layer in the architecture outputs predictions, including offsets for bounding boxes and confidence scores for each class.

SSD Training and Loss Function

The SSD loss function combines both localization and confidence losses to optimize detection accuracy:

- **Matching Algorithm**: Each anchor box is matched with a ground truth box if the IoU > 0.5, ensuring that each object is covered by at least one anchor box.

- **Localization Loss** (L_{loc}): Measures the error in bounding box position and size using Smooth L_1 loss, which is less sensitive to outliers.

- **Confidence Loss** (L_{conf}): Uses softmax and cross-entropy to penalize incorrect class predictions.

- **Total Loss**: The total loss is a weighted combination of L_{loc} and L_{conf}.

$$L(x, c, l, g) = \frac{1}{N} \left(L_{\text{conf}}(x, c) + \alpha L_{\text{loc}}(x, l, g) \right)$$

where

- N is the number of matched anchor boxes.

- L_{conf} is the softmax and cross-entropy loss over multiple confidences c.

- L_{loc} is a Smooth L1 loss between the predicted box l and the ground truth box g parameters (a regression loss).

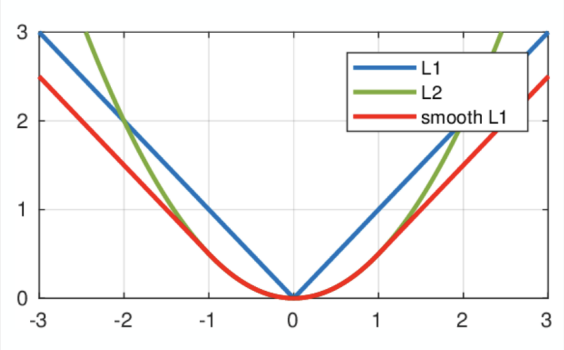


Figure 5.25: Smooth L1 loss, also known as Huber loss in some contexts

$$\text{Smooth L1 Loss}(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < \delta, \\ \delta(|x| - \frac{\delta}{2}) & \text{otherwise,} \end{cases}$$

Where:

- $x = y_{\text{pred}} - y_{\text{true}}$ is the residual (difference between prediction and ground truth).
- $\delta > 0$ is a hyperparameter determining the threshold between L_1 and L_2 behavior.

Derivative:

$$\frac{d}{dx}\text{Smooth L1 Loss}(x) = \begin{cases} x & \text{if } |x| < \delta, \\ \delta \cdot \text{sign}(x) & \text{otherwise.} \end{cases}$$

Key Points:

- For small residuals ($|x| < \delta$), the loss behaves like L_2-loss: $\frac{1}{2}x^2$, providing smoothness and higher penalty for small errors.
- For large residuals ($|x| \geq \delta$), the loss behaves like L_1-loss: $\delta(|x| - \frac{\delta}{2})$, reducing sensitivity to outliers.
- Smooth L1 loss combines the robustness of L_1 with the stability of L_2, making it differentiable everywhere.

Applications:

- **Object Detection:** Used in bounding box regression (e.g., Faster R-CNN) to handle annotation noise and outliers.
- **Regression Tasks:** Balances precision for small errors and robustness to outliers.

Hard Negative Mining

To handle class imbalance (where after the matching step, most of the default boxes are negatives - i.e. most anchor boxes are background):

- SSD performs **Hard Negative Mining** by selecting the most challenging negative samples, keeping a balanced ratio of positives to negatives.
  - Instead of using all the negative examples, we sort them using the highest confidence loss for each default box and pick the top ones so that the ratio between the negatives

and positives is at most 3:1.

- This reduces computational cost and helps focus training on difficult cases, leading to faster and more stable training.

Data Augmentation

To improve generalization, SSD applies data augmentation techniques:

- **Augmentation with Jaccard Overlap**: Include patches with a minimum overlap threshold to ensure objects are sufficiently represented.
  - generate diverse training data while ensuring the augmented data maintains a minimum degree of overlap with the original objects.
  - Generate a random crop from the original image, which may partially or completely cover the original objects.
  - Adjust the size and aspect ratio of the crop randomly within predefined limits.
  - To retain meaningful object information, only accept crops that satisfy a minimum IoU (Jaccard overlap) between the crop and at least one original bounding box.
- **Random Patches**: Introduce random variations to improve robustness.

Semantic Segmentation

Overview

- Semantic segmentation involves dividing an image into regions corresponding to different semantic classes, such as "background," "dog," and "cat."
- The objective is to classify each pixel in the image to a specific semantic category. This classification is carried out at the **pixel level**, meaning that *each pixel is assigned a label.*
- One of the widely used datasets for semantic segmentation tasks is Pascal VOC2012, which provides images with pixel-level annotations for various classes.

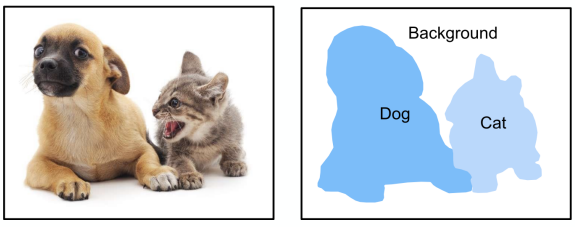


Figure 5.26: Semantic segmentation

Image Segmentation vs Instance Segmentation

Image Segmentation: This process divides an image into regions that share similar characteristics or belong to the *same semantic class*. It typically involves unsupervised or weakly supervised techniques and leverages pixel correlation to group regions.

Figure 5.27: Image segmentation

- Divides and image into constituent semantic regions.
- Exploits correlation between pixels in the image/
- Not necessarily supervised.

Instance Segmentation: Unlike semantic segmentation, instance segmentation not only identifies the class of each pixel but *also distinguishes different objects (instances) within the same class.*

For example, if there are two dogs in an image, instance segmentation would differentiate between each dog rather than treating them as a single "dog" region.

Figure 5.28: Instance segmentation

- Simultaneous detection and segmentation
- Segmentation for each object separately

Semantic Segmentation: Pixel-wise Classification

- Semantic segmentation treats image segmentation as a *pixel-wise classification problem.*
- Each pixel in the image is classified individually into classes such as "cat," "dog," or "background."

Traditional methods for this task relied on **feature engineering**, where pixel differences or local filter-based features were fed into classifiers (e.g., Random Forest) to determine the class of each pixel.

Semantic Segmentation with Deep Learning

Semantic segmentation exploits CNN feature maps.

- DL has enhanced the process of semantic segmentation by leveraging CNNs to **extract meaningful image features**.
- **Feature maps:** The output of each CNN layers, which represents a high-dimensional transformation of the input image. These feature maps capture hierarchical information about the image and serve as input for segmentation tasks.
- The deep feature maps can capture spatial details useful for segmenting regions according to semantic labels.
- The process involves passing the input image through multiple layers of convolution and pooling, resulting in multiple feature maps that capture different aspects of the image (at different levels of abstraction).
- As illustrated in the example, after applying several filters, feature maps appear to highlight certain parts of the image (e.g., cat shapes), which facilitates segmentation by showing which pixels belong to different semantic classes.

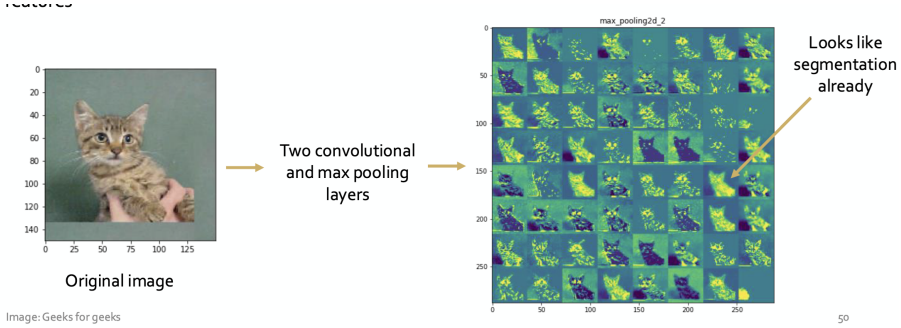


Figure 5.29: RHS> many parallel feature maps - each is a different channel into the next layer - each one is learning slightly different things

This is how semantic segmentation networks works: it's a smart trick: it exploits CNN feature maps.

Whereas CNNs take feature maps and then run prediction on them in FC layers, Semantic segmentation takes the feature maps directly??

U-Net Architecture for Semantic Segmentation

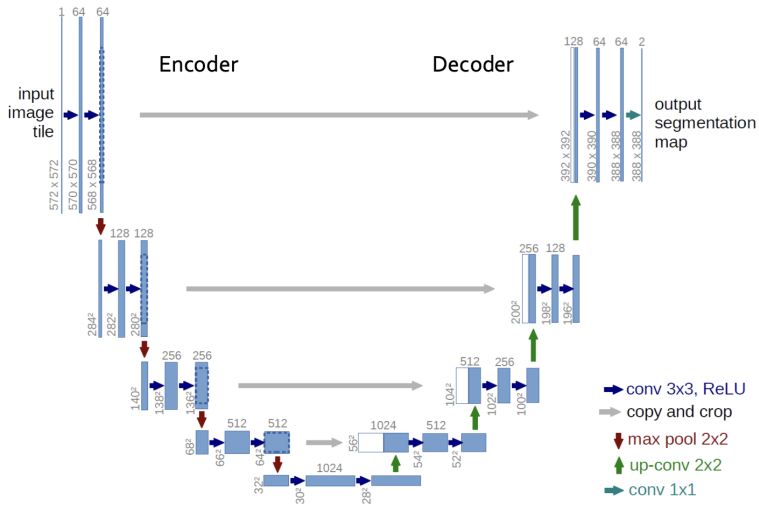


Figure 5.30: U-Net architecture

- **Architecture:** U-Net is a widely used architecture for semantic segmentation, originally developed for biomedical image segmentation.
- **Encoder-Decoder Structure:**
  - **Encoder:** Extracts features by downsampling the image to create lower-dimensional representations.
  - **Decoder:** Uses up-convolutions to upsample and reconstruct the image while assigning class labels to pixels.
- **Skip Connections:** Shortcut connections from the encoder to the decoder layers to preserve spatial information.

The architecture:

- Different levels have different feature maps at different resolutions.
- **Encoder:** transforms input data into smaller segmentation.
- **Decoder:** does? to create more examples??
- Up-convolutions: going from lower dimensional representations to higher dimensional representations

Up-Convolution / Transposed Convolution

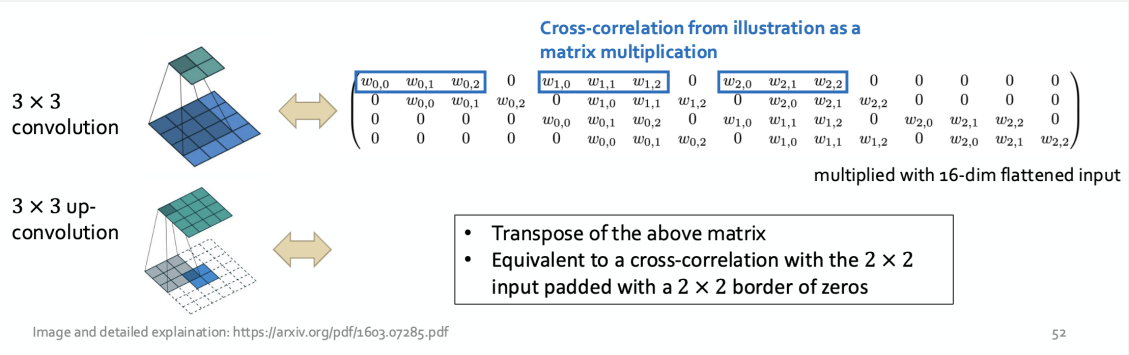


Figure 5.31: Enter Caption

- **Purpose:** Transposed convolution layers in U-Net are used to upsample feature maps in the decoder.
- **Mechanism:**
  - A transposed convolution operates in a way that is conceptually similar to a regular convolution but "reverses" the spatial dimensions.
  - The resulting feature map has a higher spatial resolution, making it ideal for reconstructing the segmented image.
- **Interpretation:** The transposed convolution layer effectively increases the resolution of the image while maintaining the learned features.

Chapter 6

Recurrent Neural Networks and Sequence Modeling

Contents

- Motivating sequences and time series in public policy
- Overview of sequence modeling tasks and challenges
- Basic building blocks that comprise state-of-the-art sequence models
  - Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU)
  - Temporal Convolutions
  - Recurrent Neural Networks (RNN)
- Example Task: Time series forecasting

6.1 Intro to Sequence Modeling

Sequential data refers to data where the **ordering of instances** matters and there are **dependencies between instances**. Unlike traditional tabular data, where each row is independent, sequential data has structure and information embedded in its order.

6.1.1 Characteristics of Sequential Data

Sequential data is characterized by:

- **Order Dependency**: The order of instances within the dataset is crucial, as rearranging them could result in loss of information or meaning.
- **Instance Dependency**: Each instance can depend on previous instances, creating a chain of dependencies across time or positions.

Examples of sequential data include:

- **Text data** where each word depends on its context.
- **Time series data** (e.g. market prices; sensor values; log files) where the values are correlated at different times.
  - Time series is sequential data that is indexed by time.

– Often (but not necessarily), time series are successive equally spaced data points indexed by time.

– Sensor data might not be equally spaced. e.g. motion sensor: go off every time someone goes by / is activated.

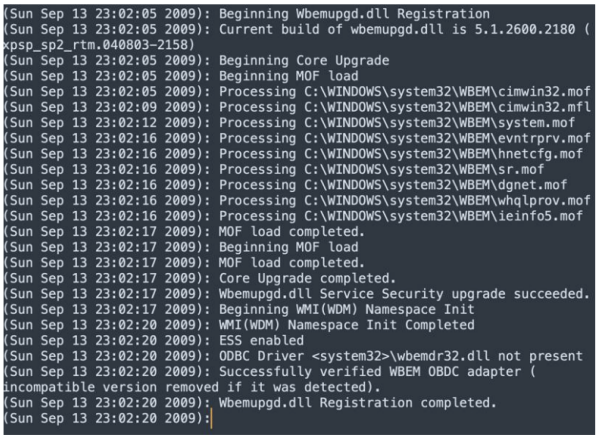


Figure 6.1: log files as time series data

- **DNA sequences** where each nucleotide's position carries meaning in the genetic code

- **Video data**

- **Sound data**

Observations for Non-Sequential Data:

1. **Order of instances within the dataset does not matter**:
   - Rearranging or shuffling the instances does not change the information content or alter the meaning of the data.
   - *Example*: In a dataset of customer records, where each row represents a different customer (e.g., age, income, location), changing the row order does not affect the information, as each record is independent.
2. **Values of one instance do not depend on values of another**:
   - Each data point or instance is independent of others, meaning the information within one row does not rely on or influence information from other rows.
   - *Example*: In an image classification dataset, each image is treated as a separate entity. The pixels in one image have no relationship or dependency on the pixels in another image.
3. **Same size of each of the instances**:
   - Non-sequential data typically has a consistent format or number of features for each instance, allowing for straightforward comparisons and modeling.
   - *Example*: In a survey dataset, each respondent has the same number of features (e.g., age, gender, response score). This fixed structure is required for traditional machine learning algorithms that expect inputs of a uniform size.

Why These Properties Matter
These characteristics simplify the modeling process for non-sequential data:

- There is no need to account for dependencies between instances, allowing models to treat each instance independently.
- Fixed-size inputs enable simpler models, as there is no requirement to handle variable-length sequences, unlike in sequential data (e.g., time series or text).

In contrast, sequential data, has **dependencies across instances**, **meaningful ordering**, and **variable length sequences**. This requires specialized models that can *capture relationships over time or positions*, fundamentally differentiating sequential from non-sequential data.

6.1.2 Challenges in Modeling Sequential Data

Modeling sequential data presents unique challenges:

- **Variable Lengths**: Unlike typical machine learning models that expect fixed-size inputs, sequential data may vary in length (e.g., sentences of varying word counts).
- **Long-Term Dependencies**: Capturing relationships that span across large time steps or positions is challenging, as information can be "forgotten" as it moves through a network.
- **Vanishing/Exploding Gradients**: During training, backpropagation can result in gradients that either vanish (become too small) or explode (become too large), especially in long sequences.

6.1.3 Sequences in Public Policy – Time Series

Time Series is a type of sequential data where each observation is *indexed by time*. This means that the order of data points is essential and carries information about temporal dependencies. In time series, data points are often equally spaced, but this is not a strict requirement. Examples of time series include:

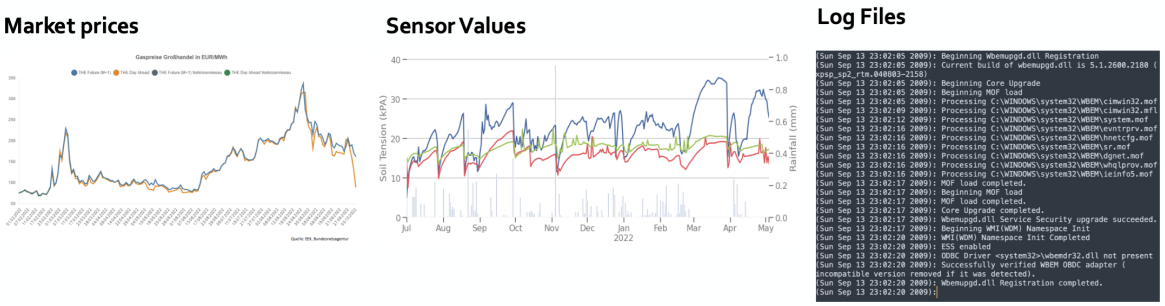


Figure 6.2: Time Series

- **Market Prices**: Financial time series, such as stock prices, show temporal trends and seasonal patterns that are crucial for forecasting.

- **Sensor Values**: Sensors record measurements over time, such as temperature or soil moisture. Analyzing these data streams can help in predictive maintenance or environmental monitoring.

- **Log Files**: System logs are recorded chronologically. Detecting unusual patterns or trends over time can reveal system errors or security breaches.

6.2 Examples of Sequence Modeling Tasks

Sequencing modeling:

These sequence modeling tasks leverage the temporal or structural dependencies within sequential data to perform a variety of predictive, diagnostic, and analytical functions.

Each task has unique challenges and requires models that can effectively capture and interpret dependencies across time steps or within subsequences.

6.2.1 Forecasting and Predicting Next Steps

Forecasting is the task of predicting future values based on past observations. In time series forecasting, models analyze patterns and dependencies in historical data to generate future estimates.

Applications include:

- **Electric Load Forecasting**: Predicting electricity consumption over time is crucial for grid management and energy planning.

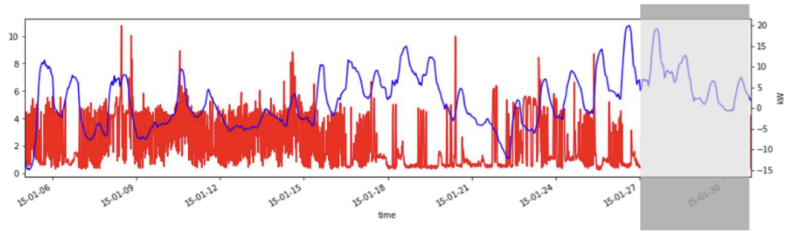


Figure 6.3: Electricity load forecasting

- **Search Query Completion**: Predictive text algorithms, such as those used in search engines, anticipate the next words in a query based on previous user inputs.

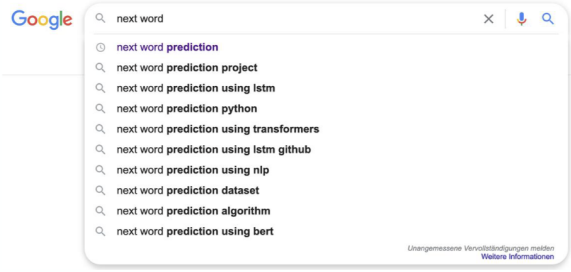


Figure 6.4: Search query completion

6.2.2 Classification

Classification tasks involve categorizing a sequence or parts of a sequence based on learned patterns.

> In sequence classification we are classifying the entire sequence into a category.

Examples include:

- **Non-Intrusive Load Monitoring (NILM)**: NILM uses energy consumption patterns from electrical devices to classify which appliances are active based on their unique power signatures.

Figure 6.5: non-intrusive load monitoring

- **Sound Classification**: Sound classification involves identifying types of sounds (e.g., engine noise, sirens, or speech) from audio recordings by analyzing frequency and amplitude patterns over time.

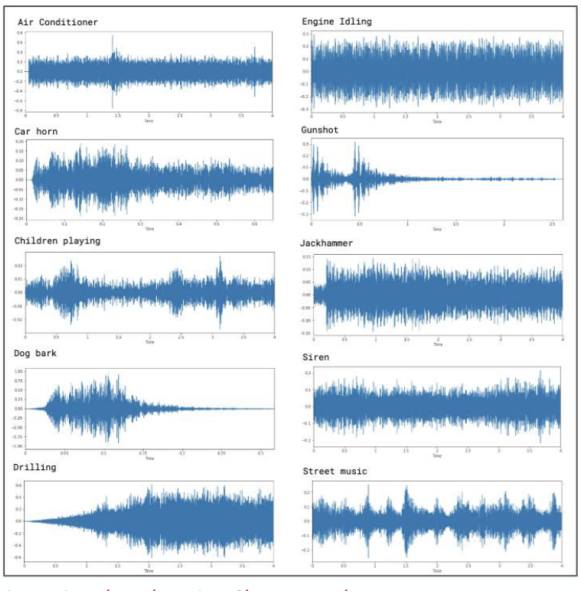


Figure 6.6: sound classification

6.2.3 Clustering

Clustering organizes sequences into groups based on similarity. This technique is useful for discovering natural groupings in data. An example in time series is as follows.

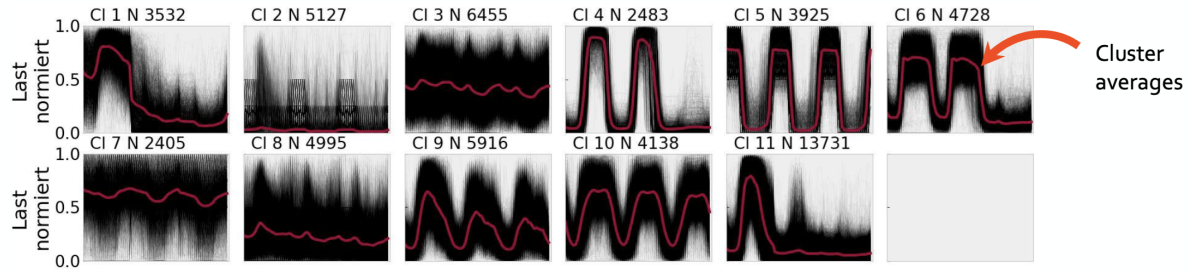


Figure 6.7: Clusters of load profiles of industrial customers determined by k-Means clustering

- **Load Profiles of Industrial Customers**: By clustering load profiles, energy companies can group customers with similar usage patterns, helping them offer tailored tariffs or demand response strategies.

- **Document classification**: where you are clustering documents into different types...

6.2.4 Pattern Matching

Finding („querying“) a known pattern within a sequence.

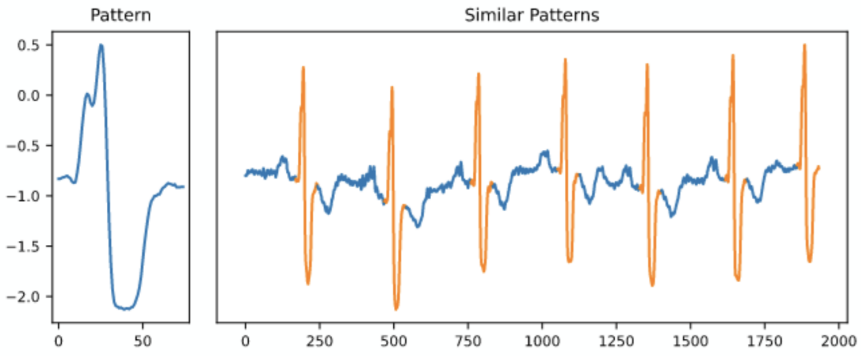


Figure 6.8: pattern matching a heartbeart

Pattern matching identifies instances of a specific pattern within a sequence. Common applications include:

- **Heartbeat Detection**: In medical data, pattern matching can locate heartbeat patterns within a continuous signal to monitor health conditions.
- **DNA Sequencing**: Finding specific DNA patterns within genetic data can help identify genes or mutations associated with diseases.

6.2.5 Anomaly Detection

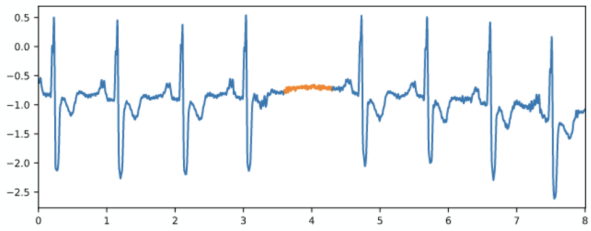


Figure 6.9: Anomaly Detection

Anomaly detection focuses on identifying unusual data points or subsequences. This is particularly useful in fields where detecting deviations from the norm is crucial, such as:

- **Predictive Maintenance**: In industrial systems, detecting anomalies in sensor readings can indicate equipment wear or imminent failure, allowing for preventative measures.

6.2.6 Motif Detection

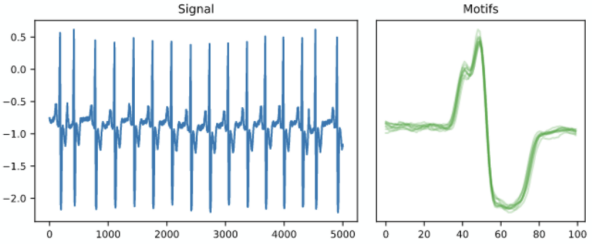


Figure 6.10: motif detection

Motif detection finds frequently occurring subsequences within a longer sequence. This is commonly used in genomic studies:

- **DNA Analysis**: Repeated patterns in DNA sequences, known as motifs, can provide insights into genetic functions or evolutionary relationships.

6.3 Sequence Modeling Techniques

Several approaches have been developed to model sequential data, each with its advantages and limitations:

6.3.1 Recurrent Neural Networks (RNN)

Recurrent Neural Networks are designed to handle sequential data by maintaining a "memory" of previous inputs. The basic idea is to pass information from one time step to the next through a *hidden state*. For an input sequence $\{x_1, x_2, \ldots, x_T\}$, an RNN calculates:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b)$$

where:

- h_t is the hidden state at time t.
- W_h and W_x are weight matrices.
- b is a bias term.
- σ is an activation function (commonly tanh).

While effective for short sequences, **vanilla RNNs** struggle with long-term dependencies due to the vanishing/exploding gradient problem.

6.3.2 Long Short-Term Memory (LSTM) Networks

LSTM networks introduce a "cell state" c_t that acts as a long-term memory, along with gates to control the flow of information:

$$\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad \text{(forget gate)} \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad \text{(input gate)} \\
\tilde{c}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad \text{(output gate)} \\
h_t &= o_t \cdot \tanh(c_t)
\end{aligned}$$

Here:

- f_t is the forget gate, deciding how much of the previous cell state c_{t-1} to retain.
- i_t is the input gate, determining how much of the new information $\tilde{c}_t$ to add to the cell state.
- o_t is the output gate, controlling how much of the cell state to output as the hidden state h_t.

LSTMs effectively handle long-term dependencies by using these gates to selectively remember or forget information.

6.3.3 Gated Recurrent Units (GRU)

GRUs are a simplified version of LSTMs, merging the forget and input gates into a single update gate z_t:

$$\begin{aligned} z_t &= \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \quad \text{(update gate)} \\ r_t &= \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \quad \text{(reset gate)} \\ \tilde{h}_t &= \tanh(W_h \cdot [r_t \cdot h_{t-1}, x_t] + b_h) \\ h_t &= (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \end{aligned}$$

GRUs require fewer parameters than LSTMs, making them faster to train and less prone to overfitting, while still capturing long-term dependencies.

6.3.4 Convolutional Neural Networks (CNNs) for Sequences

1D CNNs can also be applied to sequential data by using convolutional filters over a sliding window. For an input sequence $\{x_1, x_2, \ldots, x_T\}$, the convolutional operation is:

$$h_j = \sum_{k=-p}^{p} x_{j+k} \cdot w_{-k}$$

where w is the filter (kernel) and p is the padding size. However, CNNs generally have limited capacity to capture long-term dependencies in sequences and are commonly used in combination with RNNs.

6.3.5 Transformers and Self-Attention Mechanism

The **Transformer** model, introduced in Vaswani et al. (2017), replaces recurrence with *self-attention*, allowing each position to attend to all others in the sequence:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where:

- Q (query), K (key), and V (value) are linear transformations of the input sequence.
- d_k is the dimensionality of the keys.

Transformers excel at capturing global dependencies and are highly parallelizable, making them suitable for long sequences.

6.3.6 Choosing a Model for Sequential Data

Choosing the right model depends on:

- **Length of dependencies**: Use LSTM or GRU for moderately long dependencies, while Transformers are better for extremely long dependencies.
- **Computation constraints**: CNNs are fast, but RNNs and Transformers may be more expressive.
- **Task-specific requirements**: For tasks requiring fine-grained attention (e.g., language processing), Transformers are ideal.

Sequential Models:

Sequential data requires models that can capture dependencies across instances. Each approach offers unique benefits depending on the data characteristics and task requirements.

While RNNs were foundational, LSTMs and GRUs improved their effectiveness, and Transformers introduced a new paradigm in sequence modeling.

General Approaches to Sequence Modeling Tasks

Sequence modeling involves using sequential data to predict, classify, or detect patterns, among other tasks. These models must capture the inherent dependencies and temporal ordering within the data. There are two primary approaches to handling sequence data: **manual feature engineering** and **end-to-end learning**.

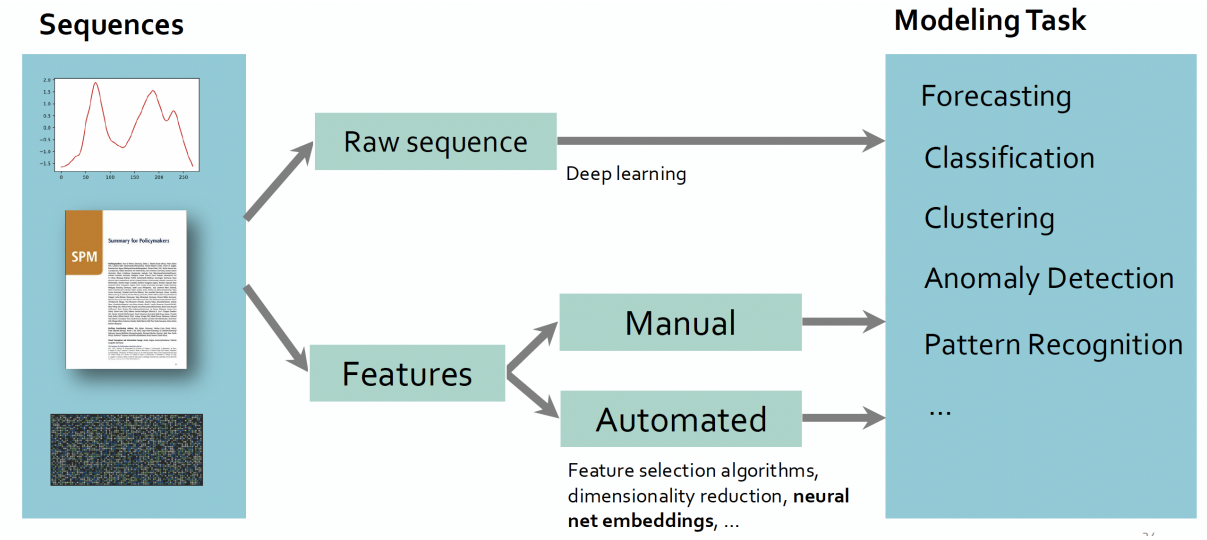


Figure 6.11: General approaches to sequence modeling tasks

DL: End-to-End Learning Vision

The vision of end-to-end learning in sequence modeling is to take the raw sequence data and *learn features and tasks directly* from it. Here feature representation is *implicit within the model.* In deep learning, this is achieved by training models on the raw sequences without manual intervention in feature extraction.

Trad ML: Feature Engineering in Sequence Modeling

In traditional machine learning, feature engineering plays a significant role, where features are manually designed or automatically generated to represent the sequence data effectively in order to improve model performance. Feature extraction can be divided into:

- **Manual Feature Engineering**: Domain experts create features based on their understanding of the data. This may include *lags, moving averages, seasonality,* and other hand-crafted patterns.

- **Automated Feature Extraction**: Using deep learning methods, features are learned automatically. Techniques like *feature selection algorithms*, *dimensionality reduction*, and *neural network embeddings* are employed to derive meaningful representations of the data without manual input.

1. Trad ML: Feature Engineering in Sequence Modeling

The feature-based approach involves first extracting features from the sequence, which are then used for modeling.

Example 1: Feature Modeling for Text Data Using Bag-of-Words

In natural language processing, a common approach for feature extraction is the **Bag-of-Words (BoW)** model. In BoW:

- Each unique word in the corpus is included in the vocabulary.
- A text sequence is represented by a vector indicating the count of each vocabulary word in the sequence.

Limitations of Bag-of-Words:

- Order of words is not preserved, we are **losing important contextual information & structure**.
- BoW can result in high-dimensional vectors as the vocabulary size increases, especially when using n-grams to capture word order.

Count Vectorization:

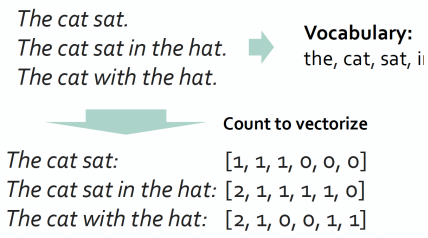


Figure 6.12: BoW

In this vectorized representation, each sequence is mapped to a fixed-length vector that indicates the occurrence of each word, but the model loses information about word order.

Example 2: Feature Modeling for Time Series Data in Load Forecasting

For load forecasting, common features include:

- **External Variables**: Weather conditions like temperature and solar irradiance.
- **Seasonality**: Patterns on daily, weekly, and annual scales.
- **Lagged Values**: Load values from previous hours or days.
- **Socioeconomic Indicators**: Data such as the number of residents, floor space, and energy tariffs.

These features are represented as variables (X) in a feature matrix and then used to predict target values (y), such as future load demands. This approach enables models to capture relationships between the features and the target variable, **but in this feature engineering approach we are still NOT exploiting the series' chronology**. Fundamentally, we could shuffle all of these examples around, and it would not matter.

To fully exploit the series aspect of our data, we will need DL...

Conclusion The choice between using raw sequences and feature-based approaches in sequence modeling depends on the complexity of the data, availability of domain knowledge, and computational resources. End-to-end learning is increasingly popular with advancements in deep learning, while feature-based methods remain useful in domains where interpretability and domain expertise are critical.

2. DL: End-to-End Learning (Raw Sequencing)

This is typically done with deep learning architectures that are designed to handle sequential data end-to-end, such as recurrent neural networks (RNNs), convolutional neural networks (CNNs) for sequences, or transformers.

Advantages:

- Models can capture complex patterns directly from the raw data without requiring predefined features.
- End-to-end learning is well-suited for tasks where the structure of the sequence itself contains valuable information.

Challenges:

- Requires large amounts of data and computational power.
- May be harder to interpret as the learned features are implicit within the network.

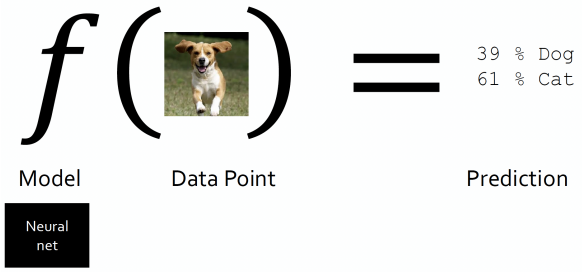
Challenges in Raw Sequence Modeling

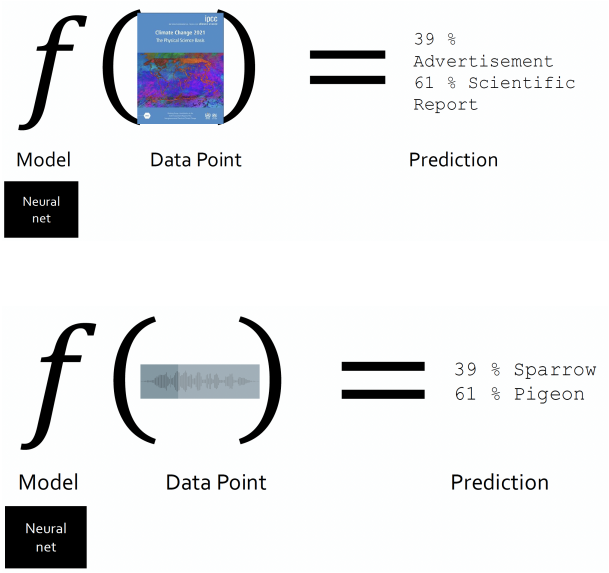
Modeling raw sequences is challenging because of the complexities inherent in sequential data.

Context: ML is function learning...

$$\underset{\text{NN model}}{f}\quad\underset{\text{Data point}}{(x)}\quad=\quad\underset{\text{Prediction}}{\hat{y}}$$

... but this gets hard for sequential data





Challenge 1: Fixed Size Requirement

$$f : R^d \rightarrow R$$

Where d is a fixed size vector. We have a short receptive field (the model will not see more than is in the filter).

Most traditional machine learning models require fixed-size inputs and outputs. However, sequence data often varies in length (e.g., sentences of different word counts, time series with variable lengths).

This limitation necessitates additional **preprocessing** or **padding** strategies when using fixed-size models.

Challenge 2: Temporal Dependencies at Multiple Scales

In many sequences, dependencies exist across both short and long time scales. For example:

- In sound processing, dependencies may exist within milliseconds (e.g., vibrations) and seconds (e.g., syllables in speech).
- In time series, dependencies may span minutes, hours, or even days, depending on the application.

This **multi-scale dependency** makes it difficult for simple models with **short receptive fields** (e.g., convolutional layers with fixed-size filters) to capture the full range of temporal patterns. Models that can learn these multi-scale dependencies, such as recurrent neural networks (RNNs) or transformers with attention mechanisms, are more suitable for such tasks.

DL modeling sequences directly from raw data without manual feature engineering is a powerful approach, but it requires handling challenges related to variable input sizes and capturing multi-scale dependencies. Techniques like RNNs, CNNs with larger receptive fields, and transformers with attention mechanisms provide solutions for these challenges, enabling more robust and flexible sequence modeling.

6.4 Sequence Models

1. Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) are a class of neural networks that excel in processing sequential data by *maintaining a connection between the elements in the sequence.*

Fully Connected Networks vs. RNNs

Traditional fully connected networks work well when the input has a **fixed dimension** d. In such networks:

- Every node in a layer is connected to every node in the next layer.
- These networks calculate a function $f(x_1, x_2, \ldots, x_d)$ where the dimension d is fixed.

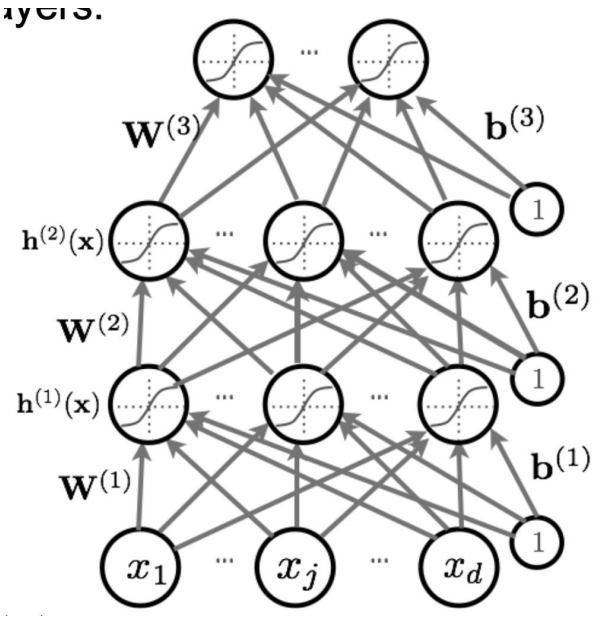


Figure 6.13: Fully Connected Network

However, when dealing with sequences of **variable length** N**,** a fully connected network is insufficient because:

- It **cannot handle inputs** of variable lengths naturally.
- It **cannot capture dependencies** between sequential elements.

1. How can we compute $f(x_1, x_2, \ldots, x_N)$ for an N that may vary?
2. How can we ensure that between the inputs there is dependence?

We calculate it *recurrently*!

The Recurrence Mechanism in RNNs

To address the challenges posed by sequential data, RNNs employ a **recurrence relation**, which allows them to process sequences of variable length while maintaining a form of "memory" of

previous elements.

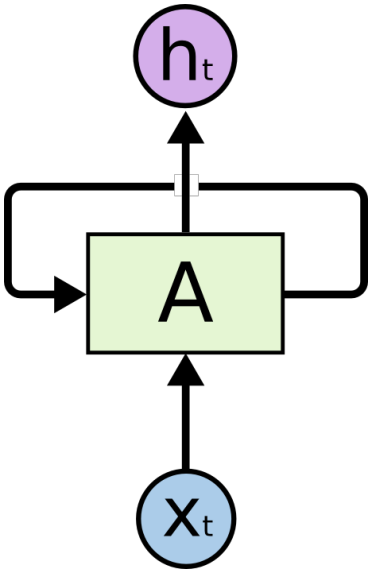


Figure 6.14: The recurrence relationship

The **recurrence** is defined as:

$$h_t = A(h_{t-1}, x_t),$$
$$\text{for } t = 1, \ldots, N$$

where:

- h_t is the **hidden state** at time step t, capturing information about the sequence up to that point.
  - h_{t-1} is the hidden state from the previous time step, which serves as a memory of prior inputs.
- x_t is the input at time step t.
- A is an activation function that combines h_{t-1} and x_t; it is a neural network function or unit, such as a simple RNN cell, LSTM, or GRU.

Typically, the activation function A is chosen to be tanh or ReLU, though tanh is more common in standard RNNs.

Time step t could be 1 token in NLP, or 1 load hour, etc.

The **final prediction** for a sequence of length N is then the **hidden state at the last time step**, denoted as:

$$f(x_1, x_2, \ldots, x_N) = h_N$$

NB this final hidden state is dependent on input from across the whole series; because of sequential dependency in the model. At each time step we feed in both the new input from the current time step and the previous time step's output from the unit.

Unrolling an RNN

RNNs can be thought of as **multiple applications of the same network** at different time steps, **each passing an activation to a successor**. This makes RNN "deep" neural networks, even if technically only one layer is modeled.

An RNN can be visualized by **unrolling** it across time steps. In this view:

- Each copy of the network at a time step **shares the same parameters and structure**.
- The hidden state h_t at each time step is passed to the next time step, enabling the network to **retain information** over time.

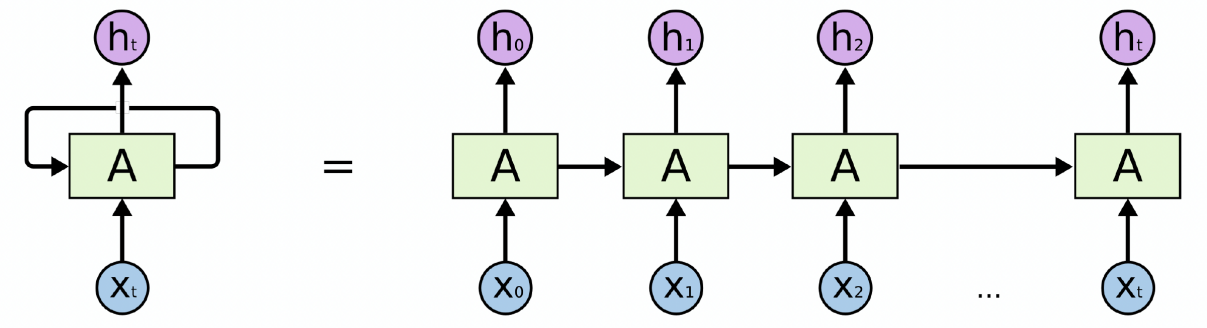


Figure 6.15: Illustration of an unrolled RNN, where h_t represents the hidden state at each time step and A denotes the RNN unit.

This unrolled representation shows that, although an RNN may consist of a single neural unit, it can be viewed as a **deep network** due to the multiple time steps.

Advantages of RNNs

- **Variable-Length Input Handling**: RNNs can process sequences of varying lengths by iterating over each element in the sequence.
- **Temporal Dependency Modeling**: RNNs maintain information about past inputs, making them effective for tasks where prior context is essential.

Challenges with RNNs

- **Vanishing/Exploding Gradients**: As the sequence length grows, gradients during backpropagation may diminish or explode, making it difficult to learn long-term dependencies.
- **Limited Long-Term Memory**: Standard RNNs struggle with retaining information over many time steps. Variants like LSTM and GRU address this issue by introducing mechanisms to control information flow.

"Vanilla" Recurrent NNs

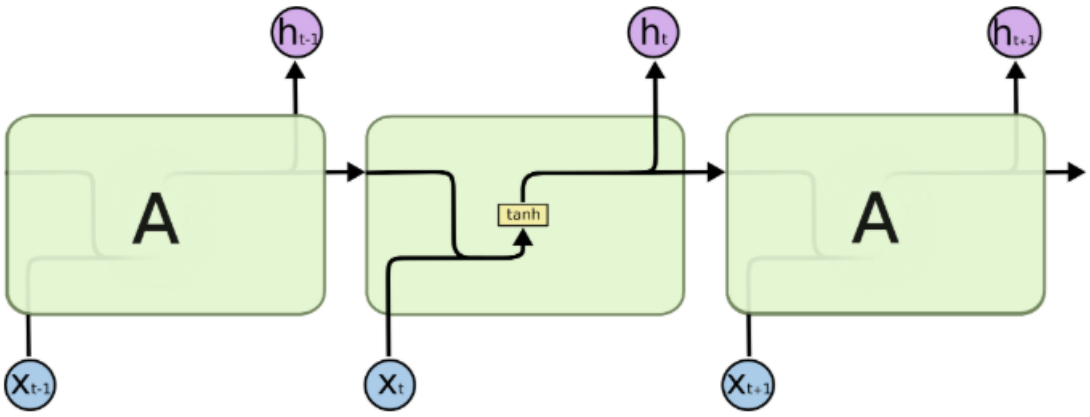
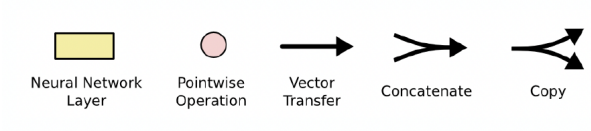


Figure 6.16: Unrolled RNN



In Vanilla RNNs, the key operation is the recurrent function that updates the hidden state h_t at each time step t:

$$h_t = A(h_{t-1}, x_t)$$

where:

- h_t is the **hidden state** at time step t, capturing information about the sequence up to that point.
  - h_{t-1} is the hidden state from the previous time step, which serves as a memory of prior inputs.
- x_t is the input at time step t.
- A is an activation function that combines h_{t-1} and x_t; it is a neural network function or unit, such as a simple RNN cell, LSTM, or GRU.

Typically, the activation function A is chosen to be tanh or ReLU, though tanh is more common in standard RNNs.

Mathematical Formulation

If A is simply an activation function like tanh, the Vanilla RNN update rule becomes:

$$h_t = \tanh(W \cdot [h_{t-1}, x_t] + b)$$

where:

- W is the weight matrix that connects the previous hidden state and the current input to the new hidden state.
- b is a bias term.
- $[h_{t-1}, x_t]$ denotes the **concatenation** of h_{t-1} and x_t.

Key Properties and Challenges

- **Short-Term Memory**: Vanilla RNNs can effectively capture dependencies *within a few time steps* but struggle with long-term dependencies due to the *vanishing gradient problem.*

- **Vanishing and Exploding Gradients**: During backpropagation, the gradients associated with earlier time steps may either vanish (become too small) or explode (grow uncontrollably), making it hard for the network to learn dependencies across long sequences.

- **Limitations for Long Sequences**: Because of these gradient issues, Vanilla RNNs are typically used for tasks with short-term dependencies and do not perform well on sequences requiring long-term memory.

Unrolled Representation

An RNN can be visualized as an unrolled network, where each time step represents a copy of the same network:

$$h_t = A(A(A(h_0, x_1), x_2), \ldots, x_t)$$

This unrolling allows us to see the chain of dependencies (... A = contains weights W a -> these compound -> vanishing/exploding gradient -> limited short term memory).(NB before with sigmoid activation fn compounding we had been dealing only with vanishing gradients, here we are dealing with any given weight matrix, so we are dealing with vanishing and/or exploding gradients.)

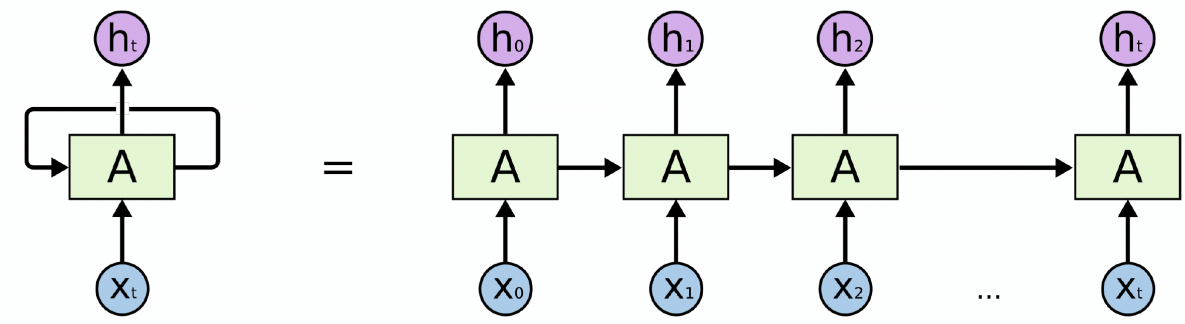


Figure 6.17: Illustration of an unrolled RNN, where h_t represents the hidden state at each time step and A denotes the RNN unit.

Parameter Sharing

Number of Weight Matrices W and Biases b:

- Weight matrices and biases are shared across each time step.
- Therefore, we have **one weight matrix** W and **one bias** b across the entire network, applied at each time step t.

Dimensions of W:

- At each time step t, the input to each RNN cell consists of the previous hidden state h_{t-1} and the current input x_t.
- Let d_h represent the size of the hidden state and d_x the number of features in the input x_t.
- Then, the dimension of W is:

$$W \in \mathbb{R}^{d_h \times (d_h + d_x)}$$

  where:

  - d_h: Size of the hidden state.
  - d_x: Number of features in the input x_t.

Concatenation of Vectors:
Let

$$h_{t-1} = [h_{t-1,1}, h_{t-1,2}, h_{t-1,3}]$$

and

$$x_t = [x_{t,1}, x_{t,2}, x_{t,3}]$$

where h_{t-1} is a vector with dimensionality d_h and x_t is a vector with dimensionality d_x (depending on input dimensionality). When we concatenate them, we get:

$$[h, x] = [h_{t-1,1}, h_{t-1,2}, h_{t-1,3}, x_{t,1}, x_{t,2}, x_{t,3}]$$

Illustrative Example of Dimensionality: $d_h = 4$ and $d_x = 3$:

$$
\begin{aligned}
h_{t-1} &= [h_{t-1,1}, h_{t-1,2}, h_{t-1,3}, h_{t-1,4}] = \mathbb{R}^4 \\
x_t &= [x_{t,1}, x_{t,2}, x_{t,3}] = \mathbb{R}^3 \\
[h, x] &= [h_{t-1,1}, h_{t-1,2}, h_{t-1,3}, h_{t-1,4}, x_{t,1}, x_{t,2}, x_{t,3}] = \mathbb{R}^7 \\
W &= \mathbb{R}^{d_h \times (d_h + d_x)} = \mathbb{R}^{4 \times (4+3)} = \mathbb{R}^{4 \times 7}.
\end{aligned}
$$

$$
\begin{aligned}
h_t &= \tanh(W \cdot [h_{t-1}, x_t] + b) \\
h_{t_{(dh \times 1)}} &= \tanh(W_{(d_h \times [d_h + d_x])} \cdot [h_{t-1}, x_t]_{((d_h + d_x)) \times 1} + b_{(d_h)}) \\
h_{t_{(4 \times 1)}} &= \tanh(W_{(4 \times 7)} \cdot [h_{t-1}, x_t]_{(7 \times 1)} + b_{(4)})
\end{aligned}
$$

Importantly:

$$
h_{t_{(4 \times 1)}} = h_{t-1_{(4 \times 1)}}
$$

Expanded Matrix View:
At each time step t, the input to an RNN cell is the concatenation of the hidden state h_{t-1} and the input x_t. For $d_h = 4$ and $d_x = 3$, the concatenated vector is:

$$[h_{t-1}, x_t] = \begin{bmatrix} h_{t-1,1} \\ h_{t-1,2} \\ h_{t-1,3} \\ h_{t-1,4} \\ x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{bmatrix}, \quad \text{with dimensions } \mathbb{R}^7.$$

The weight matrix W has dimensions $\mathbb{R}^{4\times 7}$ (as $d_h = 4$ and $d_h + d_x = 7$). It can be expanded as:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} & w_{16} & w_{17} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} & w_{26} & w_{27} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} & w_{36} & w_{37} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} & w_{46} & w_{47} \end{bmatrix}.$$

The bias vector b has dimensions $\mathbb{R}^4$ and can be written as:

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}.$$

To compute the new hidden state h_t, the RNN cell applies the following operation:

$$h_t = f(W \cdot [h_{t-1}, x_t] + b),$$

where f is a non-linear activation function (e.g., tanh or ReLU).
Expanded Computation:
The multiplication $W \cdot [h_{t-1}, x_t]$ expands as:

$$W \cdot [h_{t-1}, x_t] = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} & w_{16} & w_{17} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} & w_{26} & w_{27} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} & w_{36} & w_{37} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} & w_{46} & w_{47} \end{bmatrix} \begin{bmatrix} h_{t-1,1} \\ h_{t-1,2} \\ h_{t-1,3} \\ h_{t-1,4} \\ x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{bmatrix}.$$

This results in a vector of size $\mathbb{R}^4$:

$$W \cdot [h_{t-1}, x_t] = \begin{bmatrix} w_{11}h_{t-1,1} + w_{12}h_{t-1,2} + w_{13}h_{t-1,3} + w_{14}h_{t-1,4} + w_{15}x_{t,1} + w_{16}x_{t,2} + w_{17}x_{t,3} \\ w_{21}h_{t-1,1} + w_{22}h_{t-1,2} + w_{23}h_{t-1,3} + w_{24}h_{t-1,4} + w_{25}x_{t,1} + w_{26}x_{t,2} + w_{27}x_{t,3} \\ w_{31}h_{t-1,1} + w_{32}h_{t-1,2} + w_{33}h_{t-1,3} + w_{34}h_{t-1,4} + w_{35}x_{t,1} + w_{36}x_{t,2} + w_{37}x_{t,3} \\ w_{41}h_{t-1,1} + w_{42}h_{t-1,2} + w_{43}h_{t-1,3} + w_{44}h_{t-1,4} + w_{45}x_{t,1} + w_{46}x_{t,2} + w_{47}x_{t,3} \end{bmatrix}.$$

Adding the bias vector b gives:

$$W \cdot [h_{t-1}, x_t] + b = \begin{bmatrix} (\text{row } 1) + b_1 \\ (\text{row } 2) + b_2 \\ (\text{row } 3) + b_3 \\ (\text{row } 4) + b_4 \end{bmatrix}.$$

Finally, the activation function f is applied element-wise to produce the updated hidden state h_t:

$$h_t = f\left(\begin{bmatrix} (\text{row } 1) + b_1 \\ (\text{row } 2) + b_2 \end{bmatrix} \right)$$

Explanation of h_{t-1} as a Vector and Vector Concatenation
In the context of Recurrent Neural Networks (RNNs), h_{t-1} represents the **hidden state** from the previous time step. This hidden state is a vector because it contains a set of values (elements) that represent the internal memory of the RNN at time $t-1$. These values are crucial for maintaining temporal information across time steps.
Let's denote:

$$h_{t-1} = \begin{bmatrix} h_{t-1,1} \\ h_{t-1,2} \\ h_{t-1,3} \end{bmatrix}$$

where h_{t-1} has dimensionality $d_h = 3$ in this example. Similarly, let the input vector at time t be:

$$x_t = \begin{bmatrix} x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{bmatrix}$$

where x_t has dimensionality $d_x = 3$.

Concatenation of h_{t-1} and x_t To feed both the previous hidden state and the current input into the RNN at time t, we concatenate h_{t-1} and x_t into a single vector:

$$[h_{t-1}, x_t] = \begin{bmatrix} h_{t-1,1} \\ h_{t-1,2} \\ h_{t-1,3} \\ x_{t,1} \\ x_{t,2} \\ x_{t,3} \end{bmatrix}$$

Visual Representation of Dimensions If h_{t-1} has dimensionality d_h and x_t has dimensionality d_x, then the concatenated vector $[h_{t-1}, x_t]$ will have dimensionality $d_h + d_x$.

$$\text{Dimensionality of } h_{t-1} = \big[d_h\big] = \big[3\big]$$

$$\text{Dimensionality of } x_t = \big[d_x\big] = \big[3\big]$$

$$\text{Dimensionality of } [h_{t-1}, x_t] = \big[d_h + d_x\big] = \big[6\big]$$

Thus, the concatenation operation stacks the elements of h_{t-1} and x_t into a single vector with a combined dimensionality of $d_h + d_x$.

Short term memory

Standard RNN can model short-term contexts easily.

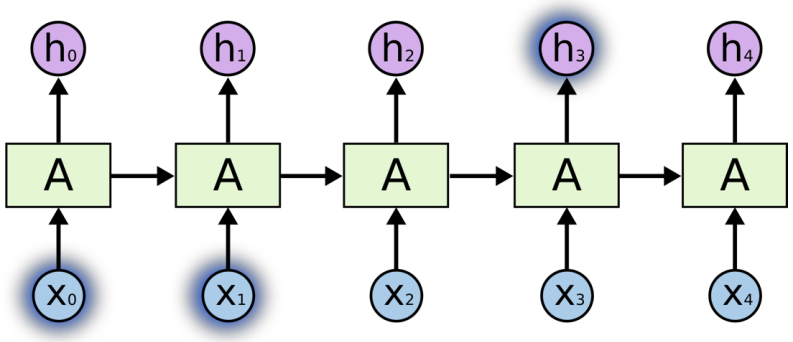


Figure 6.18: Short term context: "the clouds are in the *sky*"

However, they struggle for sequences, the model becomes relatively deep.

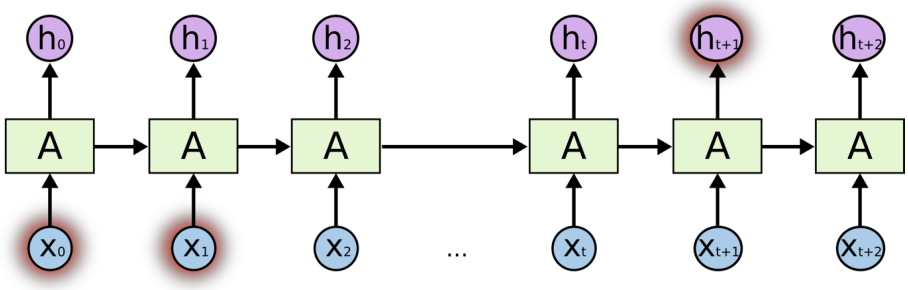


Figure 6.19: Long term context: "I grew up in France... I speak fluent *French*."

This is because of the vanishing/exploding gradient problem.

Consider only 3 steps:

$$h_3 = A(A(A(h_0, x_1), x_2), x_3)$$

Each A here, contains a W term; lots of weights compounding each other!

Due to the vanishing/exploding gradient problem, until recently we were stuck with short term memory - the only way to avoid vanishing/exploding gradients whas through memorylessness.

Therefore, never really worked well for practical applications (fell into the "**AI winter**").

> Before we had only ever been dealing with vanishing gradients derived from saturation of the sigmoid function.
>
> Here we are talking about both vanishing and exploding gradients.

NLP Application

For example, in a language processing task:

- Given a sequence of words, the RNN processes each word one at a time.
- At each step, it updates its hidden state based on the current word and the previous state, allowing it to build a contextual understanding.

However, Vanilla RNNs can typically only capture short-term dependencies (e.g., a few words) and may fail to understand broader context in long sentences.

Excursion I: Output layers and vector notation

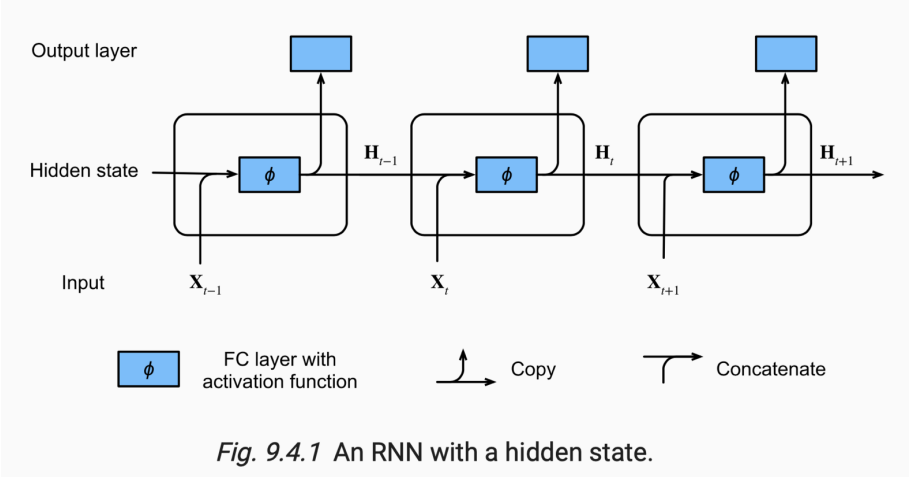


Fig. 9.4.1 An RNN with a hidden state.

Figure 6.20: RNN with a hidden state

- **Hidden State**:
  - A hidden state is the hidden layer output of dimensions $n \times h$,
  - Each row in H_t represents the hidden state for an individual sequence in the batch at time step t
  - it represents an internal state vector that maintains information about previous inputs in a sequence.

$$R \in n \times h$$

A hidden state is the output of the hidden layer at a specific time step in a Recurrent Neural Network (RNN). Its dimensions are $n \times h$, where:

- n: The batch size, representing the number of sequences being processed in parallel.
- h: The size of the hidden state vector, often denoted as the number of hidden units in the RNN.

Each row in H_t represents the hidden state for an individual sequence in the batch at time step t. Specifically:

$$H_t = \begin{bmatrix} h_{t,1} \\ h_{t,2} \\ \vdots \\ h_{t,n} \end{bmatrix}, \quad \text{where each row } h_{t,i} \in \mathbb{R}^h.$$

The hidden state represents an internal state vector that encodes information about the sequence observed up to time step t. It helps maintain a memory of previous inputs in the sequence, enabling the RNN to model dependencies over time.

The dimensions of the hidden state matrix H_t are:

$$H_t \in \mathbb{R}^{n \times h}.$$

- **Hidden Layers in Fully Connected Neural Networks**: In a traditional fully connected neural network, the hidden layer can be expressed as follows:

$$\begin{aligned} H^{[2]} &= \phi(H^{[1]} W^{[2]}) \\ &= \phi(\phi(X W^{[1]}) W^{[2]}) \end{aligned}$$

  This structure is typical for standard neural networks where each layer processes information **independently** of previous layers in a sequence.

- **Hidden State in RNNs**: In an RNN, we aim to process a sequence of inputs while preserving information across time steps. The hidden state at each time step t, denoted H_t, captures the **cumulative** information up to that time step. Mathematically, the hidden state in an RNN is updated as:

$$\begin{aligned} H_t &= A(X_t, H_{t-1}) \\ &= \phi(X_t W_{xh} + H_{t-1} W_{hh} + b_h) \end{aligned}$$

  where:

  - W_{xh} is the weight matrix connecting the input X_t to the hidden state,
  - W_{hh} is the weight matrix connecting the previous hidden state H_{t-1} to the current hidden state,
  - ϕ is the activation function, often tanh or ReLU.

Hidden state in RNN $H_t \in \mathbb{R}^{n \times h}$

Hidden states can be used for predictions in the output layer

- **Output Layer in RNNs**: The output at each time step t, denoted O_t, is generated from the hidden state:

$$O_t = H_t W_{hq} + b_q$$

where:

  - W_{hq} is the weight matrix from the hidden state to the output layer,
  - b_q is the bias term for the output layer.

This output layer can be tailored to generate different types of predictions, depending on the task (e.g., classification, regression).

Excursion II: Backpropagation Through Time (BPTT)

BPTT is a training method used to optimize RNNs by applying the chain rule of calculus through each time step in the sequence. The key difference from standard backpropagation is that BPTT unfolds the RNN across time, treating each time step as a layer in a "deep" network.

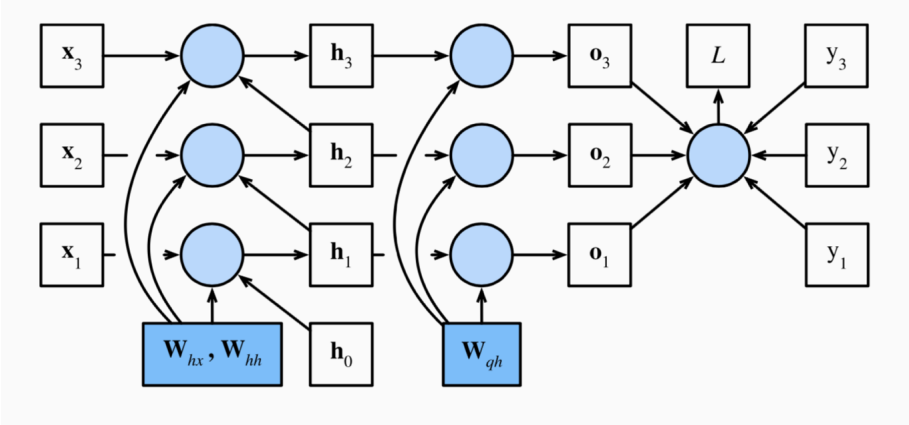


Figure 6.21: Computational graph showing dependencies for an RNN model with three time steps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators

Exploding/vanishing gradients:

Here we can see the loss L depends on the ys and the os (activations), which are themselves dependent on the previous hidden states (hs), which are v dependent on 2 repeated weight matrices W. These weight matrices compound as we go along the sequence.

- **Hidden State and Output in RNNs**: For each time step t, the hidden state h_t and output o_t are computed as:

$$h_t = W_{hx} x_t + W_{hh} h_{t-1}$$

$$o_t = W_{qh} h_t$$

where:

  - W_{hx} maps the input x_t to the hidden state,

  - W_{hh} is the recurrent weight matrix that links the hidden state at $t-1$ to the current hidden state at t,
  - W_{qh} maps the hidden state to the output.

- **Unrolling the RNN**: To understand the dependencies across time, we unroll the RNN, treating each time step as a separate layer in the network. For example, a sequence of three time steps would have hidden states h_1, h_2, and h_3, each dependent on the previous one.

- **Gradient Computation in BPTT**: Due to the recursive dependency, gradients with respect to the loss function L involve multiple applications of the weight matrix W_{hh} through the chain rule. The gradient of the loss with respect to the hidden state at time t, h_t, is:

$$\frac{\partial L}{\partial h_t} = \sum_{i=t}^{T} \left(W_{hh}^{Transp}\right)^{T-i} W_{qh}^{Transp} \frac{\partial L}{\partial o_{T+t-i}}$$

  where T is the total number of time steps. Each successive application of W_{hh} can lead to:

  - **Vanishing Gradients**: If W_{hh} has eigenvalues less than one, the gradient norms shrink exponentially over time steps, making it difficult for the model to learn long-term dependencies.
  - **Exploding Gradients**: If W_{hh} has eigenvalues greater than one, the gradients grow exponentially, causing instability during training.

  These issues make training standard RNNs challenging for sequences with long-term dependencies, often necessitating gradient clipping or alternative architectures such as LSTMs or GRUs.

This caused *AI winter*, and is the motivation for the following models.

2. Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU)

$$\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad \text{(forget gate)} \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad \text{(input gate)} \\
\tilde{c}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad \text{(output gate)} \\
h_t &= o_t \cdot \tanh(c_t)
\end{aligned}$$

NB: f_t, i_t, $\tilde{c}_t$, o_t all linear combinations of a $(W \cdot [h_{t-1}, x_t] + b)$

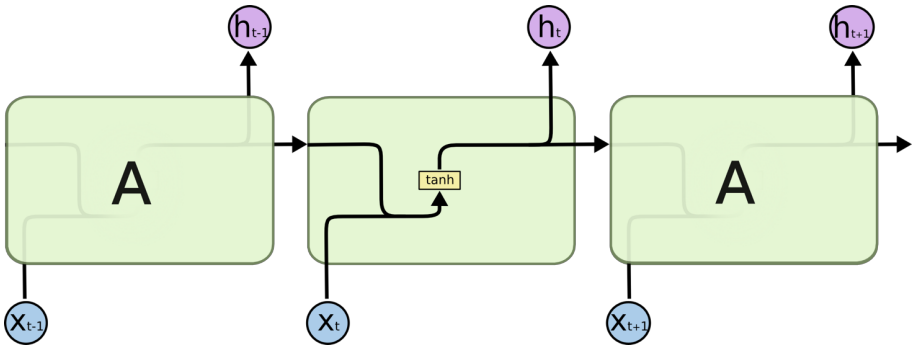


Figure 6.22: Vanilla RNN

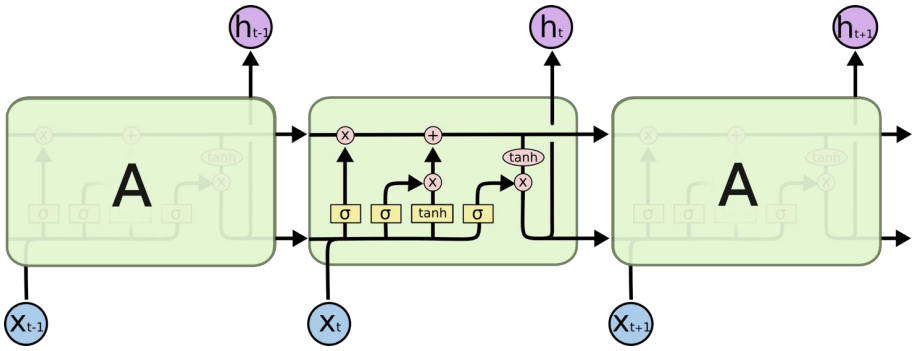


Figure 6.23: LTSM

LSTMs are a specialized form of Recurrent Neural Networks (RNNs) designed to handle the problem of long-term dependencies. They mitigate issues like the vanishing and exploding gradient problem by incorporating a **gating mechanism**.

1. Cell State (c_t) and Hidden State (h_t)

The LSTM **maintains two states** for each time step: the *cell state* c_t and the *hidden state* h_t.

Cell State: acts as a *memory carrier*, maintaining long-term information over many time steps with *minimal modifications* (no weights). (= **long term memory / across cells**)

Hidden State: is the *output* for each time step, containing the *recent information* from the sequence that is relevant to the current computation. (= **short term memory / cell output**)

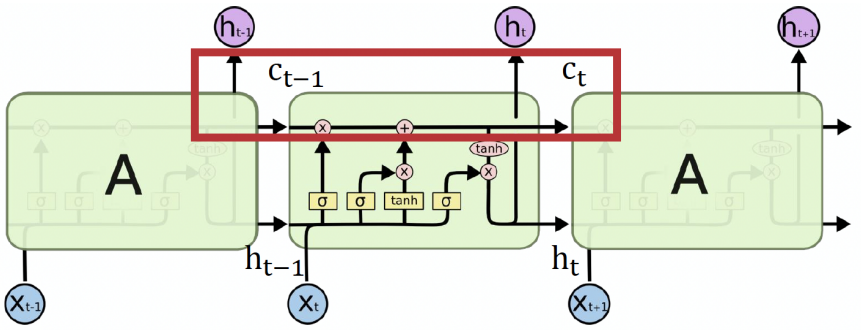


Figure 6.24: Cell state

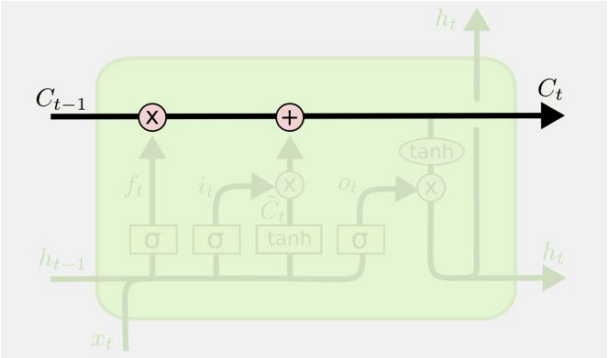


Figure 6.25: Cell state

Now the **current state (both h_t and c_t)** depends on:

1. current input value x_t,
2. previous hidden state h_{t-1}
3. previous cell state c_{t-1}

2. Forget Gate

Overview:

The forget gate controls how much information from the previous cell state c_{t-1} should be retained.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

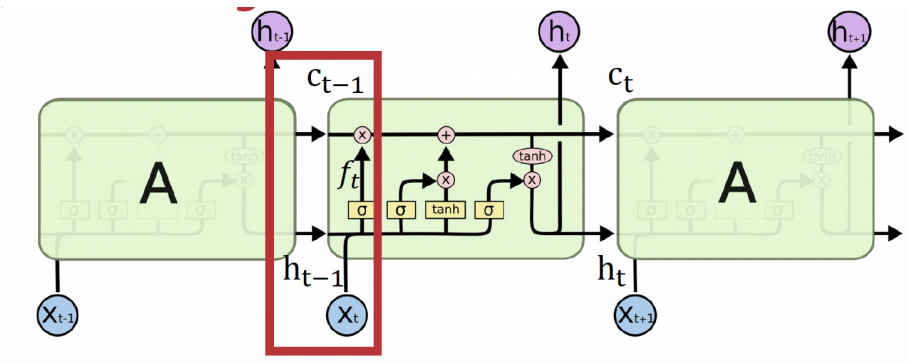
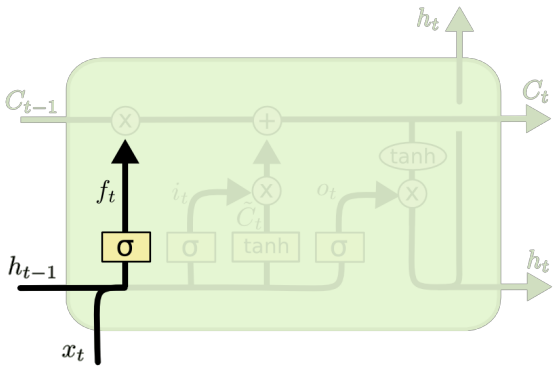


Figure 6.26: Forget gate



This is achieved through a **sigmoid** activation function:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where W_f and b_f are the weights and bias for the forget gate.

The sigmoid functional form of the forget gate outputs a value between 0 and 1:

- 0: completely forget the information (of the preceding cell state),
- 1: retain everything.

Forget gate's relation to previous hidden state and cell State:

The forget gate relies on the previous hidden state h_{t-1} (combined with x_t) to determine what information should be kept or forgotten in the cell state c_{t-1}. Specifically:

- The previous hidden state h_{t-1} provides context about prior inputs and is combined with the current input x_t to influence f_t.
- The resulting forget vector f_t then acts element-wise on c_{t-1}, modulating it to produce an updated cell state c_t.

Updated Cell State Computation:
After calculating f_t, the cell state is updated as follows:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

where:

- i_t is the input gate, which controls how much new information $\tilde{c}_t$ should be added to the cell state,
- $\tilde{c}_t$ is the candidate cell state, representing new information computed from h_{t-1} and x_t,
- $\odot$ represents element-wise multiplication.

In this way, the forget gate f_t and the input gate i_t jointly control the balance between retaining old information and integrating new information into the cell state c_t.

The forget gate:

1. Determines how much of the previous cell state c_{t-1} is retained in c_t,
2. Uses the previous hidden state h_{t-1} (along with x_t) to compute this forget ratio f_t,
3. Ensures that the cell state can maintain long-term dependencies by selectively discarding irrelevant information, allowing the LSTM to focus on the most pertinent information as the sequence progresses.

3. Input Gate

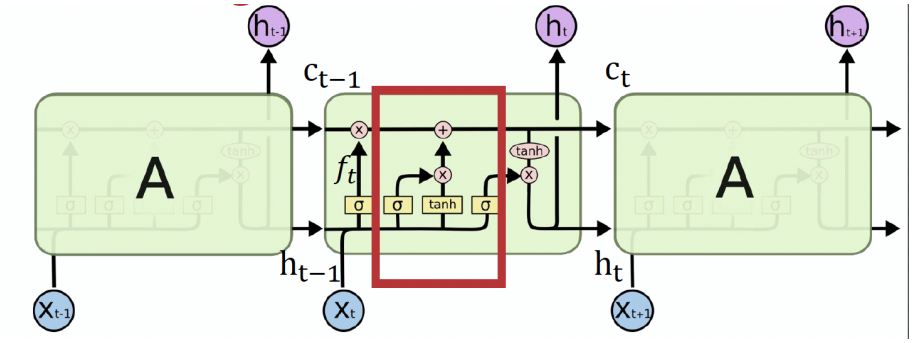


Figure 6.27: Input gate

Overview:

The input gate determines what / how much of the new information (i.e. $\tilde{c}_t$ - itself derived from the current input and the previous hidden state) should be added to the cell state for the next time step.

- relies on previous hidden state h_{t-1} and current input x_t
- affects current cell state c_t

It consists of two components:

- **Input Gate Layer:** Determines how much of the new information to add:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

- **Candidate Cell State:** Creates a candidate for the new information to be added:

$$\tilde{c}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

The cell state is updated as:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t$$

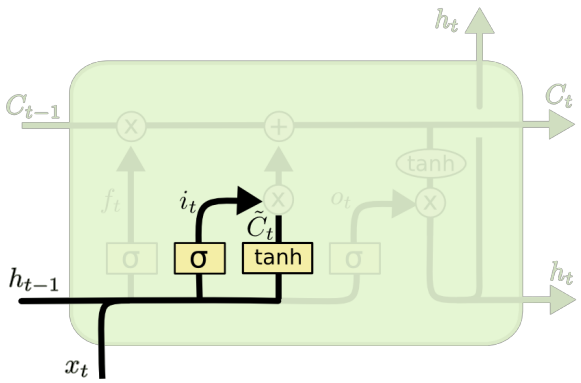


Figure 6.28: Input gate

It consists of two components:

1. **Input Gate Layer/Activation (i_t): Determines how much of the new information to add.**

   Specifically, how much of the candidate cell state $\tilde{c}_t$ should be added to the current cell state.

   $$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

   where:

   - σ is the sigmoid activation function, which outputs values between 0 and 1.
   - W_i is the weight matrix associated with the input gate.
   - h_{t-1} is the previous hidden state, providing context from prior inputs.
   - x_t is the current input to the LSTM cell.
   - b_i is the bias term for the input gate.

   The sigmoid function ensures that i_t acts as a gating mechanism, where values close to 1 allow more information from $\tilde{c}_t$ to pass through, while values close to 0 restrict it. This means the input gate activation i_t essentially acts as a "filter" to decide how much of the new information is relevant to add to the cell state.

2. **Candidate Cell State ($\tilde{c}_t$):** Creates a candidate for the new information to be added.

   Represents the new information itself that can *potentially* be added to the cell state.

   $$\tilde{c}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

   where:

   - tanh is the hyperbolic tangent function, producing values between -1 and 1.
   - W_C is the weight matrix associated with the candidate cell state.
   - h_{t-1} and x_t are the previous hidden state and current input, respectively, as in the input gate activation.
   - b_C is the bias term for the candidate cell state.

The candidate cell state $\tilde{c}_t$ represents new information generated based on the current input and the prior context. This information is scaled by the input gate activation i_t to control the degree to which it influences the overall cell state.

Bringing it all together...

Updating the Cell State with the Input Gate:

(As before) the cell state is updated as:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

where $\odot$ denotes element-wise multiplication.

The cell state c_t at time step t is updated by combining (1) the retained information from the previous cell state c_{t-1} (modulated by the forget gate f_t), (2) the new information $\tilde{c}_t$ (modulated by the input gate i_t).

This equation demonstrates the core functionality of the input gate:

- The input gate activation i_t determines the amount of the candidate cell state $\tilde{c}_t$ that should be added to c_t.

- By filtering $\tilde{c}_t$ through i_t, the LSTM decides whether to allow or restrict new information from influencing the cell state.

Input gate

- Acts as a filter to decide how much of the candidate cell state $\tilde{c}_t$ should be added to the cell state c_t.
- Modulates the influence of new information on the cell state based on the current input x_t and previous hidden state h_{t-1}.
- Allows the LSTM to manage long-term dependencies effectively by controlling the integration of new information over time.

4. Output Gate

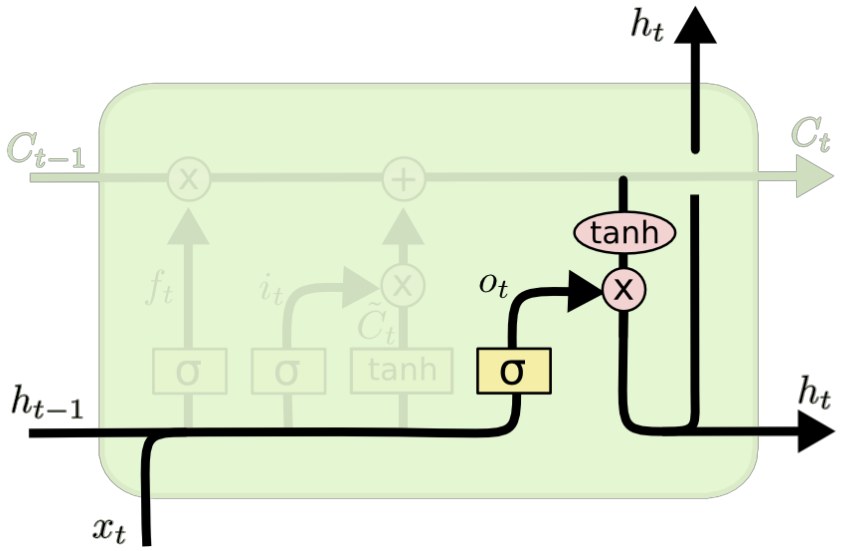


Figure 6.29: Output gate

Overview:

The output gate decides what information from the cell state should be passed on as the hidden state.

It uses the cell state to generate the hidden state:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \cdot \tanh(c_t)$$

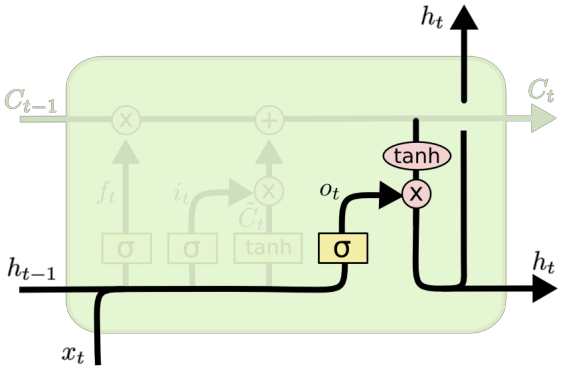


Figure 6.30: Output gate

The **output gate** is responsible for determining the content of the hidden state h_t at each time step, which ultimately serves as the output of the LSTM cell and is propagated to the next layer or time step.

This gate regulates *how much of the information stored in the cell state c_t should be exposed* and used in the current time step, balancing the model's need to provide relevant short-term

information while preserving long-term dependencies.

The output gate operates based on two main components:

- **Output Gate Activation** o_t: Controls the extent to which the cell state c_t is revealed to the next layer or time step by modulating the hidden state h_t.
- **Modulated Cell State** $\tanh(c_t)$: The cell state c_t is first passed through a tanh function to compress its values to a range between -1 and 1, enabling controlled, smooth adjustments by the output gate activation o_t.

1. **Output Gate Activation** o_t:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

where:

- σ is the sigmoid activation function, which outputs values between 0 and 1.
- W_o is the weight matrix for the output gate.
- h_{t-1} is the previous hidden state, representing prior context.
- x_t is the current input to the LSTM cell.
- b_o is the bias term for the output gate.

Similar to other gates, the sigmoid function in o_t enables a gating mechanism, where values close to 1 allow more of the cell state information to pass through, while values close to 0 restrict it.

2. **Computing the Hidden State** h_t:

$$h_t = o_t \odot \tanh(c_t)$$

where:

- $\odot$ denotes element-wise multiplication.
- $\tanh(c_t)$ transforms the cell state values to lie within the range [-1, 1], allowing smoother transitions in the hidden state.
- o_t modulates $\tanh(c_t)$ to control the amount of information from the cell state that is exposed as the hidden state.

In this equation, the hidden state h_t is formed by applying the output gate o_t to the modulated cell state $\tanh(c_t)$. This enables the LSTM to control the visibility of the information in c_t while maintaining essential long-term information in the cell state for future time steps.

Intuitive Explanation

The output gate serves as a filter for the information stored in the cell state c_t, determining what portion of this information should be shared with other parts of the network (such as the next layer or the next time step). By regulating the hidden state h_t based on both the cell state and the current input, the output gate ensures that:

- The LSTM can selectively reveal only relevant aspects of the cell state at each time step, adapting to the needs of the specific prediction or task.

- The model can balance long-term information (stored in c_t) and short-term, context-sensitive information (controlled through o_t), enhancing the ability to manage dependencies across varying timescales.

- By maintaining a controlled flow of information, the output gate helps prevent the model from becoming overwhelmed by unnecessary details, thus preserving meaningful information across the sequence.

Summary of the Output Gate's Role

- **Filter Function**: The output gate decides how much of the cell state c_t should influence the current hidden state h_t, thereby determining what information should be output to the next time step.

- **Balancing Information**: The output gate modulates the trade-off between preserving long-term information in c_t and providing relevant, immediate information through h_t.

- **Control Mechanism**: By applying σ on o_t and tanh on c_t, the LSTM can maintain smooth, stable control over the hidden state, enhancing the model's capacity to retain or forget information as required.

2.5 Quick Gated Recurrent Units (GRU) Walkthrough

GRUs are similar to LSTMs but are computationally simpler and often perform comparably for tasks involving sequences.

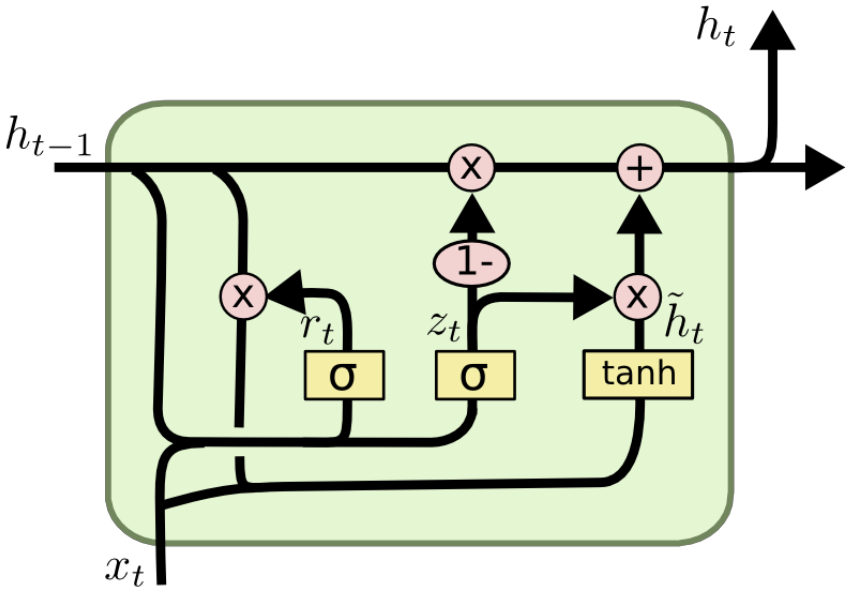


Figure 6.31: GRU unit

The GRU combines

1. the forget and input gates into a single *update gate*,

2. the cell and hidden states into a single state.

This leads to fewer parameters and a simpler structure.

It is simpler than the LSTM, and is hence faster to train and less prone to overfitting.

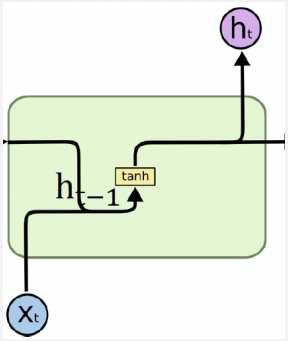


Figure 6.32: Vanilla RNN unit.png

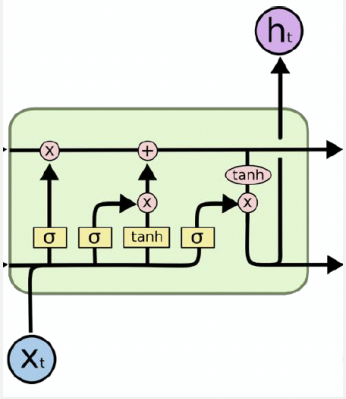


Figure 6.33: LSTM unit

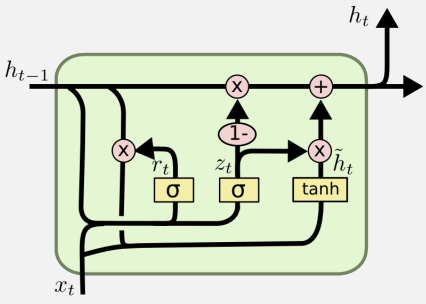


Figure 6.34: GRU unit

1. Update Gate z_t

The update gate controls how much of the previous hidden state is retained.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

The update gate decides how much of the previous hidden state h_{t-1} should be retained and how much of the new candidate hidden state $\tilde{h}_t$ should be added.

The update gate plays a role similar to the combination of the forget and input gates in an LSTM, but it combines both functions in a simpler manner.

2. Reset Gate

The reset gate determines how much of the previous hidden state to forget. This is especially useful when the model needs to reset its memory for new sequences or after a long dependency:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

When r_t is close to 0, it effectively "resets" much of the previous hidden state, allowing the GRU to focus on the new input.

3. Candidate Hidden State

The candidate hidden state $\tilde{h}_t$ represents a new potential hidden state based on the reset gate's filtering of h_{t-1}.

It is generated based on the reset gate. It allows the GRU to decide which parts of the previous hidden state are relevant:

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)$$

where:

- tanh is the hyperbolic tangent activation function, squashing the values to a range between -1 and 1.
- $\odot$ denotes element-wise multiplication.

By modulating the contribution of h_{t-1} through r_t, the reset gate allows the GRU to selectively forget past information when computing the candidate hidden state.

4. Final Hidden State

The final hidden state h_t is a blend (interpolated) of the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$, controlled by the update gate z_t.

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

This equation combines the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$ in a weighted manner based on z_t.

- When z_t is close to 1: the GRU prioritizes the candidate hidden state $\tilde{h}_t$, effectively updating its memory with new information.
- When z_t is close to 0: the GRU favors retaining the previous hidden state h_{t-1}, preserving past information.

- The LSTM uses 3 gates (forget, input, and output) and maintains two states (c_t and h_t) for long and short-term memory management.
- The GRU maintains 2 gates (update, reset), and a single state (h_t), making it simpler and faster to train.
  - update = forget + input

Intuitive Explanation

- The **update gate** z_t determines how much of the previous hidden state is retained versus how much of the new candidate hidden state is incorporated. This gate allows the GRU to decide when to "update" its memory with new information.
- The **reset gate** r_t controls how much of the past hidden state should contribute to the calculation of the candidate hidden state, enabling the GRU to "reset" or "forget" past information when it is irrelevant to the current input.

By having only two gates instead of three (like in LSTMs), GRUs are computationally lighter and faster to train.

Limitations of LSTM and GRU Models

- **Training Difficulty:** LSTMs and GRUs can be challenging to train effectively. They are prone to overfitting, especially in time series data, where capturing fine-grained patterns over extended sequences can lead to a model that does not generalize well.
- **Depth Issues:** For practical applications, these models can get extremely deep. For instance, processing a sequence of 100 words in NLP means passing through 100 layers, which increases the computational burden and complicates training.
- **Slow Training:** LSTMs and GRUs are slow to train because they are not easily parallelizable. The sequential nature of their design means that each time step relies on the computations from previous time steps, limiting the scope for parallel processing.
- **Limited Transfer Learning:** Unlike models like transformers, LSTMs and GRUs have not shown significant success in transfer learning. Adapting pre-trained LSTMs for new tasks is challenging, and they require substantial retraining for different datasets or domains.

As a result:

Popularity Decline: While LSTMs were once very popular, especially in the field of NLP, they have been increasingly replaced by transformer-based architectures. Transformers can model long-term dependencies more effectively, handle large datasets with attention mechanisms, and leverage transfer learning more efficiently.

Continued Use Despite Transformer Success: Despite being outperformed by transformers in many areas, LSTMs and GRUs are still used in certain applications, particularly where computational resources are limited or for tasks that do not require handling very long-term dependencies.

3. Convolutional Neural Networks for Sequence Modeling

Overview of CNNs for Sequence Data

Convolutional Neural Networks (CNNs) can be effectively used for sequential data by processing input through **sliding windows** of data points. In this approach, the input sequence is divided into smaller overlapping "windows" or segments, which are then passed through convolutional layers. This enables CNNs to capture local dependencies within each window, making them

suitable for tasks like time-series forecasting, language modeling, and other sequential prediction problems.

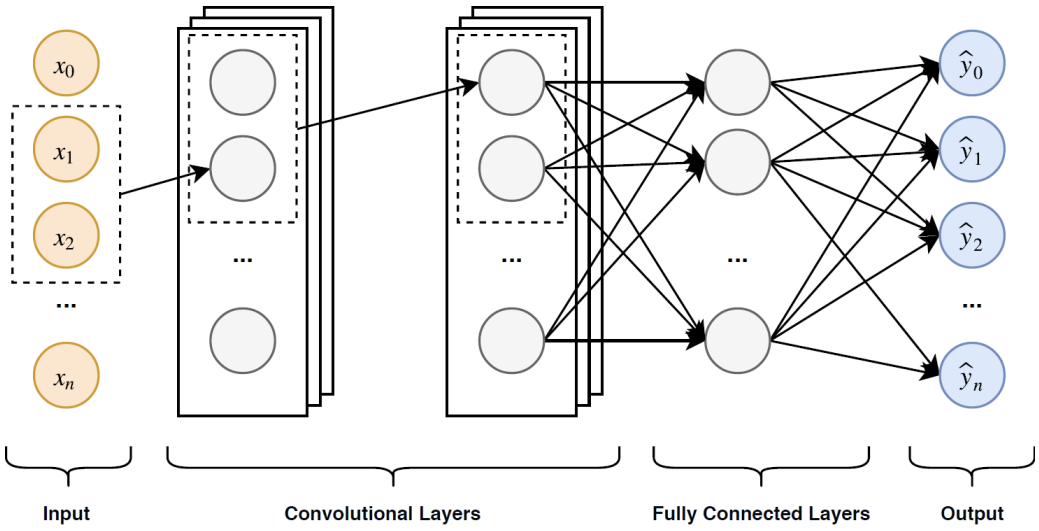


Figure 6.35: CNN can be used for sequential input by feeding "windows" of data.

1D Convolutions

1D convolutions are commonly applied to sequential data by convolving a filter (or kernel) across the sequence. For an input sequence $x = [x_1, x_2, \ldots, x_n]$ and a kernel $w = [w_{-p}, \ldots, w_0, \ldots, w_p]$ of size $2p + 1$, the convolution operation to compute output h_j at position j can be defined as:

$$h_j = \sum_{k=-p}^{p} x_{j+k} \cdot w_{-k}$$

This operation enables each output h_j to represent **locally weighted sum of neighboring input values**, capturing local patterns and dependencies.

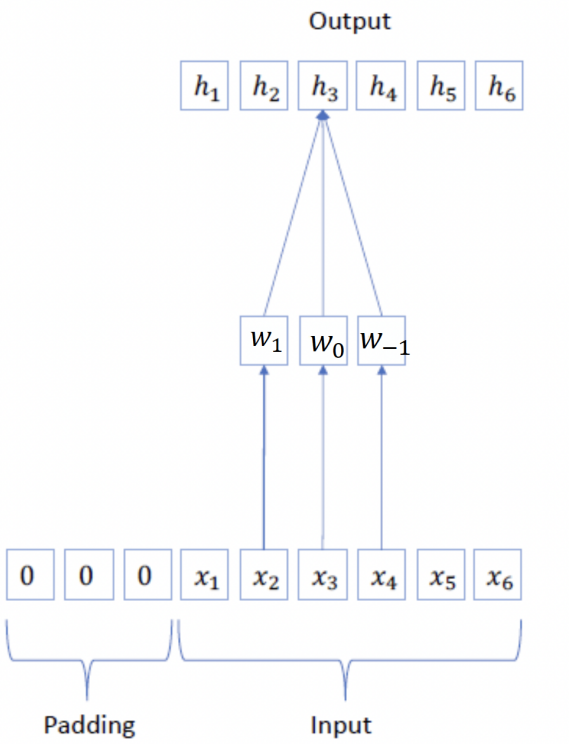


Figure 6.36: Enter Caption

Simplified to cross correlations:

Again, we can flip the order of the kernel:

Operation becomes: $h_j = (x_{j-1} \times w_{-1}) + (x_j \times w) + (x_{j+1} \times w_1)$

						x_1	x_2	x	x_4	
						1	3	3	0	

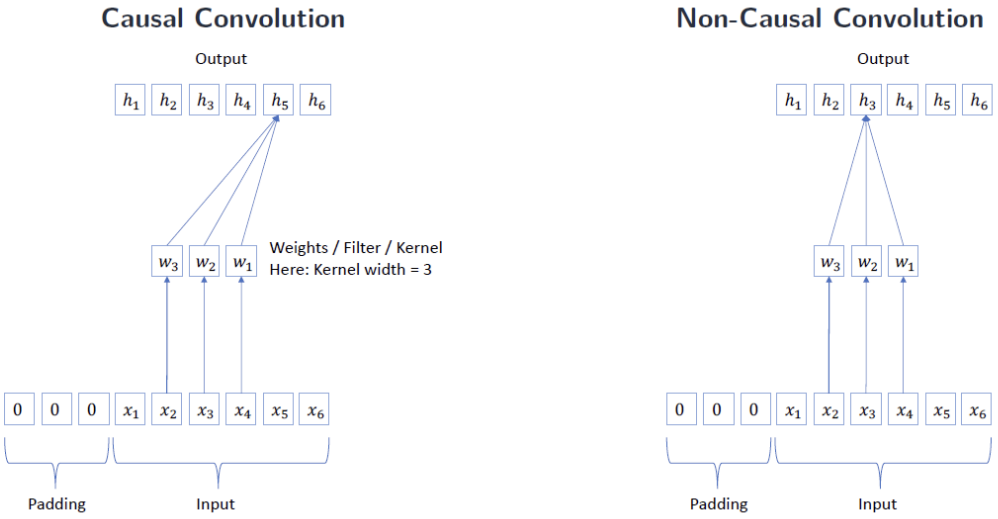
Causal Convolutions

One limitation of regular convolutions in sequence modeling is that they may introduce **future data leakage**, where the model's predictions at each timestep are influenced by future values.

To avoid this, **causal convolutions** are used, which only involve past and current input values when computing each output. In causal convolutions, for output h_j, the convolution is defined as:

$$h_j = \sum_{k=0}^{p} x_{j-k} \cdot w_k$$

Here, only past and current values contribute to the output at each timestep, making causal convolutions appropriate for applications that require strictly temporal dependencies without future information.



Dilated Convolutions

Another challenge with CNNs for sequence modeling is the **limited receptive field**.

Standard convolution layers only capture a small, fixed-range context within each layer, which may be insufficient for tasks requiring long-term dependencies.

Dilated convolutions address this by "dilating" or "spacing out" the kernel elements, enabling a larger receptive field without increasing the number of layers.

For a dilation factor d, a dilated convolution computes h_j as:

$$h_j = \sum_{k=-p}^{p} x_{j+d \cdot k} \cdot w_{-k}$$

By increasing the dilation factor across layers (e.g., $d = 1, 2, 4, \ldots$), the model exponentially expands its receptive field, allowing it to capture long-range dependencies in the sequence data.

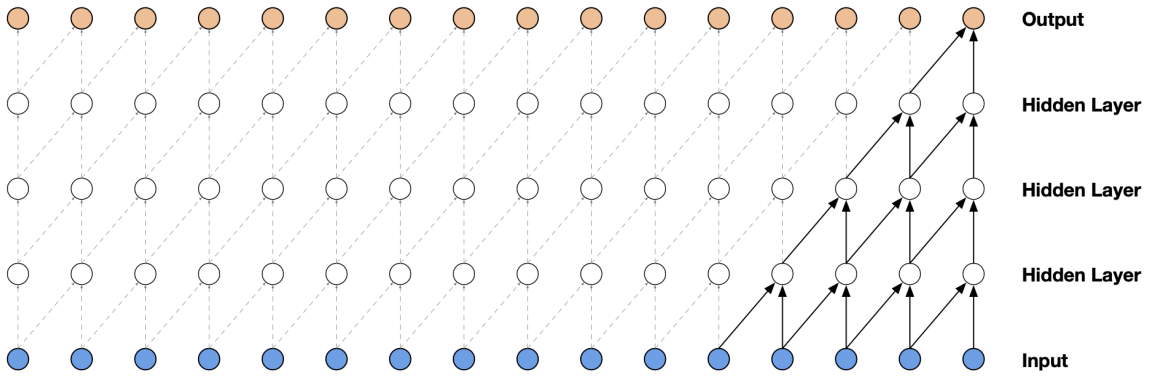


Figure 6.37: Causal Convolutions

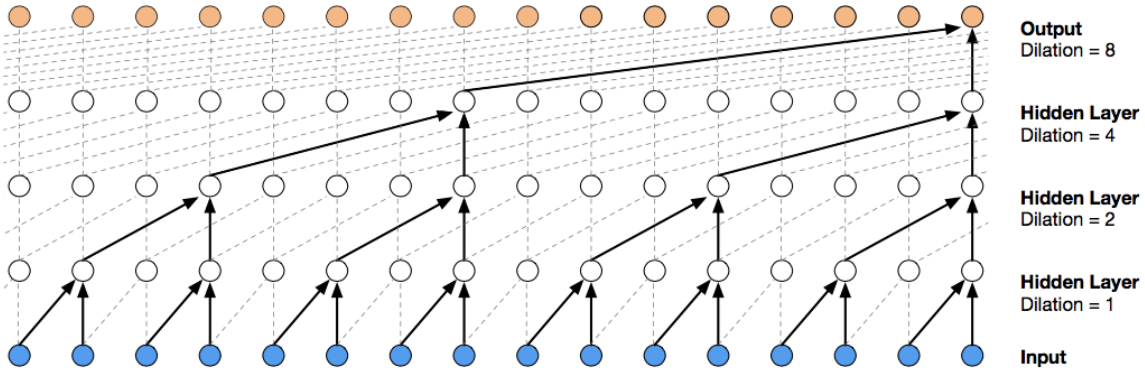


Figure 6.38: Dilated Convolutions

Limitations and Challenges of CNNs for Sequences

Strengths compared to Recurrent Neural Networks (RNNs):

- **Parallelization:** CNNs can be parallelized, making them much faster than RNNs for sequence processing tasks. This parallelization is possible because CNNs can process multiple input data points simultaneously, unlike RNNs, which generally process one step at a time.
- **Vectorisation:** more vector operations in CNNs -> faster.

Weaknesses:

- **Not Truly Sequential:** CNNs are not inherently designed to handle sequential dependencies, as they lack mechanisms to preserve temporal order across different time steps.
- **Fixed Input Length Requirement:** CNNs require fixed-length inputs, which limits their flexibility with variable-length sequences. Although padding can be used to address this limitation by standardizing input lengths, this approach introduces extra computation and may reduce performance due to the added "blank" information.

...

- **Future Leakage in Non-Causal Convolutions**: Non-causal convolutions introduce future data into the model, which can result in unrealistic performance on prediction tasks where future information should not be accessible.
- **Limited Receptive Field in Standard Convolutions**: Standard convolutions have a fixed receptive field, making it challenging to capture long-term dependencies. Solutions such as dilated convolutions help expand the receptive field but add complexity to the model.

By addressing these issues, CNNs can be adapted for sequence modeling tasks, providing an alternative to recurrent models like LSTMs and GRUs. However, the fixed receptive field and need for causal operations in time-dependent tasks remain important considerations when designing CNN-based sequence models.

4. Transformers

See Lecture 8 for Transformers.

Transformers represent a powerful model architecture that has revolutionized the field of sequence processing and NLP. Crucially they can be much more receptive in terms of what other parts of the sequence matter in their predictions than RNNs or CNNs.

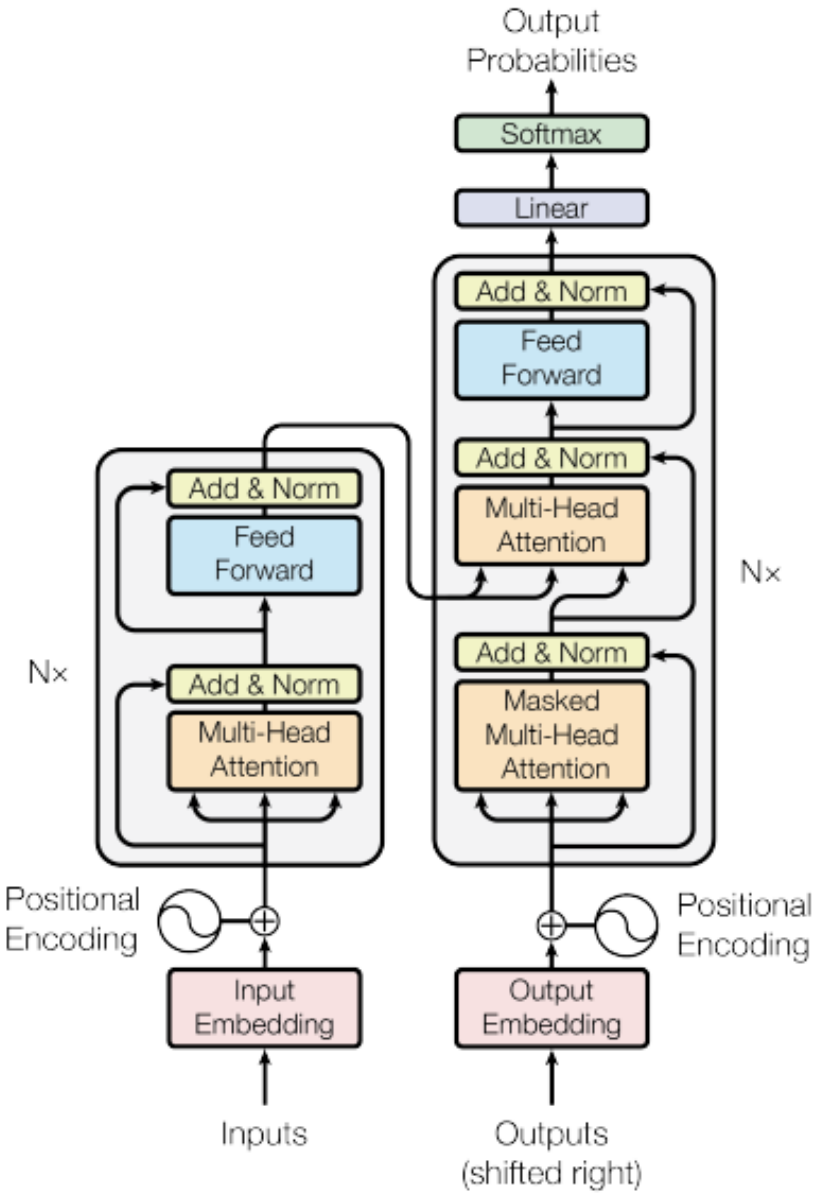


Figure 6.39: Transformer Architecture

- **Architecture and Functionality:** Transformers are fundamentally different from traditional recurrent neural networks (RNNs) and long short-term memory networks (LSTMs). While RNNs and LSTMs rely on sequential data processing, Transformers use an **attention mechanism** to process all input data *simultaneously*, making it *parallelizable* and significantly faster. This architecture allows Transformers to excel in capturing long-range dependencies across entire sequences, overcoming the limitations of RNNs and LSTMs.

- **Self-Attention Mechanism:** At the heart of the Transformer is the **self-attention mechanism**. Self-attention enables the model to weigh the importance of different tokens in the input sequence relative to each other. For each input token, the self-attention mechanism calculates a *weighted representation of all tokens in the sequence*, allowing the

model to understand the context around each word. This is crucial for tasks that require understanding dependencies, such as language translation or sentiment analysis.

- **Parallelization and Scaling:** Unlike RNNs and LSTMs, which process tokens sequentially, Transformers process all tokens in a sequence *simultaneously*. This parallelization makes Transformers much more computationally *efficient* and *scalable*. The self-attention mechanism can handle dependencies at different positions without waiting for previous tokens, enabling faster training and inference, particularly on large datasets.
- **Positional Encoding:** Since Transformers do not inherently account for the sequential nature of data, they use **positional encodings** to inject information about the position of each token in the sequence. These encodings are added to the input embeddings and provide a sense of order, which is necessary for handling sequential data effectively.
- **Multi-Head Attention:** To further enhance the model's ability to capture relationships at different levels of granularity, Transformers employ **multi-head attention**. This mechanism involves multiple attention heads, each focusing on different aspects or positions within the sequence. The results from these heads are then concatenated and linearly transformed, allowing the model to jointly attend to information from different representation subspaces.
- **Feed-Forward Layers and Residual Connections:** After the multi-head attention, each layer in the Transformer includes a **feed-forward network** that applies additional transformations to enhance non-linearity and expressiveness. **Residual connections** and **layer normalization** are also applied throughout the model to stabilize training and help with gradient flow, allowing for deeper architectures.

Transformers have demonstrated remarkable performance in NLP and sequence modeling tasks, and their scalability has led to their widespread adoption across various domains. The attention mechanism allows them to handle long-range dependencies, making them superior to LSTMs and GRUs in many contexts.

6.5 Example Task: Time Series Forecasting

WaveNet and Temporal Convolutional Networks (TCN)

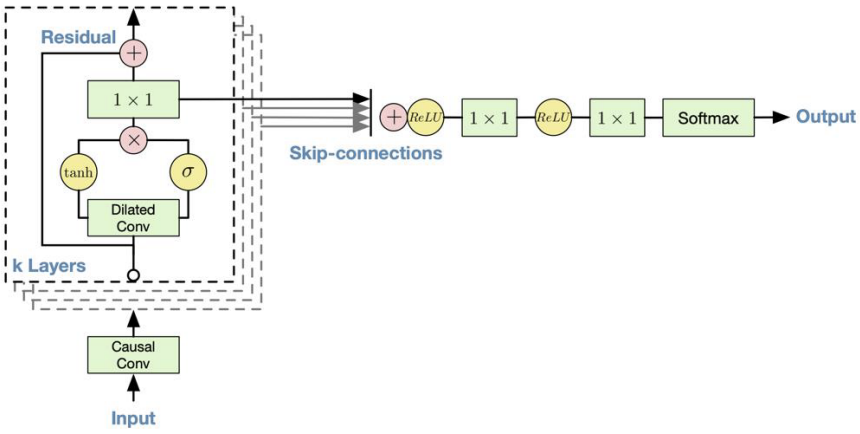


Figure 6.40: WaveNet

WaveNet:

- WaveNet, developed by DeepMind, was originally designed for generating high-frequency data such as audio signals.
- Its architecture leverages dilated and causal convolutions to model long-range dependencies without recurrent connections.
- By stacking multiple layers of dilated convolutions, WaveNet can handle high-resolution temporal data, making it effective for generating realistic sounds, such as speech and music.

Temporal Convolutional Networks (TCN):

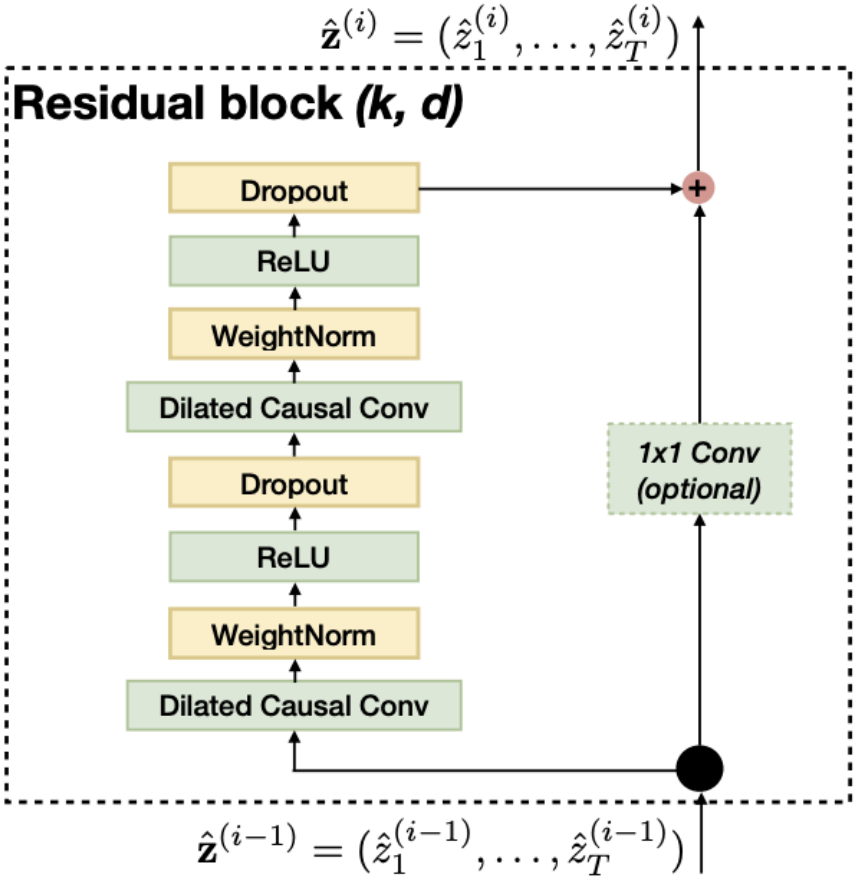


Figure 6.41: TCN

- TCNs generalize the WaveNet architecture and apply it to broader time-series and sequence modeling tasks.
- TCNs rely on key concepts:
  - **Dilated Convolutions:** These allow for an exponentially growing receptive field, making it possible to capture long-range dependencies without deep recursion.
  - **Causal Convolutions:** Ensures that each output at time t only depends on inputs from time steps $\leq t$, which is crucial for time-series tasks where future data should not influence past states.
  - **Residual Connections and Regularization:** Inspired by ResNet, TCNs employ residual connections to facilitate training deep networks, along with dropout for regularization.

- These modifications enable TCNs to handle complex temporal dependencies effectively without recurrence, making them efficient for a variety of time-series applications.

Should You Use Deep Learning for Time Series Forecasting?

Historically, time-series forecasting has primarily relied on **statistical models**, such as:

- Autoregressive models like AR, ARMA, ARIMA, and ARIMAX, which extend the basic autoregressive model to handle moving average components and exogenous variables.
- Exponential smoothing models, including Holt-Winters for handling seasonality, and the Theta method for capturing trends and seasonality.

However, **machine learning** approaches, particularly deep learning, have become increasingly popular for specific tasks:

- In cases where data is large, complex, or involves multiple interacting series, such as multivariate time-series.
- When modeling complex dependencies, non-linear relationships, and probabilistic forecasting, deep learning models can outperform traditional methods.

When are statistical models preferred?

- For simpler, *local models* where a model is fit for specific instances like a single product or location.
- When *data resolution is low* (daily, weekly, or yearly).
  - Anything that is annual in measurement: ML performs poorly. AI/ML performs v badly on macro economic indicators
- When seasonality and external covariates are *well understood.*

When are deep learning models preferred?

- For complex *global models* that need to capture dependencies across multiple related time series.
  - e.g. Speech recognition - you want a model that performs well on *all* voices, not just one.
- For *hierarchical* time-series data (i.e. when multiple units contribute to an aggregate time series).
- For *probabilistic forecasting* with *complex densities* (e.g. multimodal distributions) is required.
  - probabilistic forecast different from a point forecast; it's better to predict the entire distribution of potential values - gives you a sense of errors etc
  - But, you are predicting a whole random variable it's much harder than point forecasts.
- For applications with *complex, non-linear* external covariates and interactions, irregular seasonalities, etc where statistical methods may struggle.

Approach to Time Series Forecasting with Deep Learning

In this work flow - we are required to show that our simple models are being out competed by more complex DL models.

- **Start with Benchmark Models:** Begin with simpler, interpretable models to establish baseline accuracy.
  - seasonal models,
  - Regression models
  - ARIMA, exponential smoothing, Theta
- **Start with Feature (Engineering) Modeling:** Focus on understanding key features in the dataset and consider using advanced tabular models like random forests or gradient boosting as intermediate steps.
- **Implement Deep Learning:** After establishing benchmarks and understanding key features, implement deep learning models and compare them against traditional benchmarks.

By incrementally increasing model complexity, practitioners can build robust forecasting systems that blend interpretability and predictive accuracy.

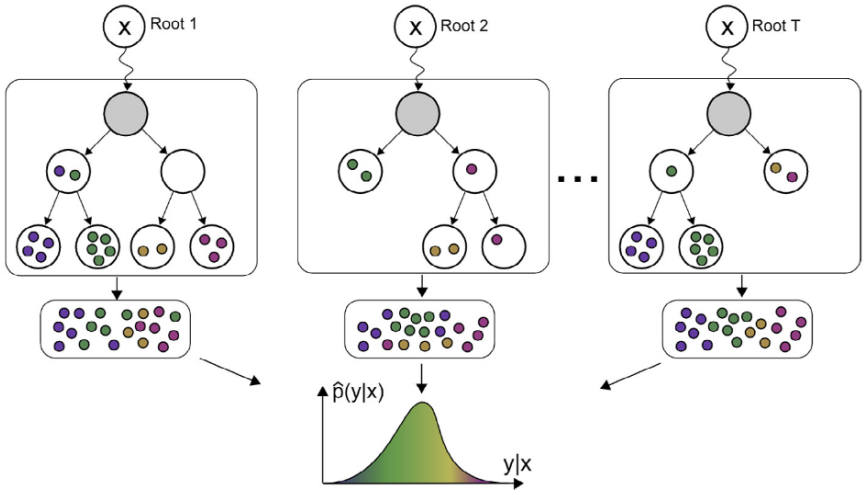


Figure 6.42: Tree Ensembles for Time Series Prediction

Chapter 7

Natural Language Processing I

7.1 Introduction

Text and Public Policy

Language, as a primary form of human communication, holds a central role in political and legislative contexts. Political discourse is predominantly text-based, encompassing a variety of sources including:

- **Legislative texts**: These include formal documents such as laws and regulations.
- **Parliamentary records and speeches**: Textual records of discussions, debates, and speeches in legislative bodies.
- **Party manifestos**: Political parties outline their goals, policies, and positions on various issues in manifesto documents.
- **Social media**: Platforms like X (formerly Twitter) generate vast amounts of political content in real-time, often reflecting public sentiment and policy discussions.

Traditional Text Analysis in Political Science: Political science has a long-standing tradition of analyzing text manually by categorizing or **"coding"** it to draw insights. This approach, however, is labor-intensive and struggles to keep pace with the increasing volume of textual data generated today.

Shift to Automated Text Analysis *at scale*: Modern techniques leverage *deep learning* and *text-as-data* methodologies, which allow for large-scale automated analysis. Deep learning models can process extensive datasets, identifying patterns and extracting meaningful information, making them a powerful tool for analyzing text in the context of public policy.

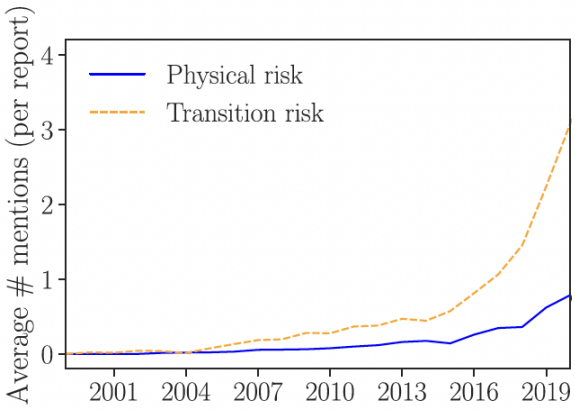
Example: Analyzing Party Manifestos and Climate Risk Disclosure

1. **Analyzing Party Manifestos (Bilbao-Jayo and Almeida, 2018)**:
   - The *Manifesto Project* involves annotating election manifestos across multiple categories to classify political party positions.
   - **Categories**: Manifestos are organized into 56 categories across seven key policy areas, such as *external relations*, *freedom and democracy*, *political systems*, *economy*, and *social groups*.
   - **Method**: Sentence classification is performed using *convolutional neural networks (CNNs)* across several languages (e.g., Spanish, Finnish, German).

- **Challenges**: One challenge is the linguistic diversity of political texts; not all languages are adequately represented in available machine learning models.

> **NLP's Equity Issue:**
>
> Most languages are not natively represented by ML models.

2. **Identifying Climate Risk Disclosure (Friedrich et al., 2021)**:
   - **Dataset**: This study analyzes a corpus of over 5000 corporate annual reports, focusing on paragraphs that discuss climate-related financial risks.
   - **Relevance**: Identifying such disclosures assists in understanding companies' risk exposure and informs investment and policy decisions.
   - **Methodology**: Document classification is performed using the *BERT* transformer model, which is effective at contextualizing and classifying text at scale.



Common NLP Tasks in Deep Learning

- **Text Classification**: Categorizing text into predefined classes, such as:
  - *Sentence and Document Classification*: Labeling sentences or entire documents by topic or sentiment.
  - *Sentiment Analysis*: Determining the emotional tone of the text (e.g., positive, negative).
  - *Natural Language Inference*: Matching text to other text statements; assessing logical relationships between sentences (e.g., entailment or contradiction).
- **Question Answering**: Extracting or generating answers to questions based on given context.
- **Named Entity Recognition (NER)**: Identifying entities such as names, locations, and dates within text.
- **Text Summarization**: Creating concise summaries from longer texts.
- **Machine Translation**: Translating text from one language to another.
- **Dialogue Management**: Supporting human-computer interaction by understanding and generating conversational responses.

Text as Data

To analyze text as data, several core concepts are used:

- **String**: A sequence of characters that represents the text.
- **Tokens**: Units derived from the text, such as words or characters, which serve as the basic elements for processing.
- **Sequence**: Tokens are sequentially organized to convey meaning.
- **Corpus**: A collection of documents, where each document is a separate text entity.
- **Vocabulary**: The set of unique tokens present in the corpus.
- **n-grams**: Contiguous sequences of n tokens, used to capture contextual dependencies.
- **Embedding**: The process of transforming text into numerical vectors, enabling machine learning algorithms to process and analyze it.

7.2 Document Embedding

Bag of Words

The most rudimentary vectorisation of documents.

Figure 7.1: BoW

	the	red	dog	cat	eats	food
1. the red dog →	1	1	1	0	0	0
2. cat eats dog →	0	0	1	1	1	0
3. dog eats food →	0	0	1	0	1	1
4. red cat eats →	0	1	0	1	1	0

Figure 7.2: Word Occurrence Matrix / Count vectorisation

- **Word Order Ignorance**: BoW ignores the sequence of words and treats the document as an unordered collection of words. This means that sentences like "the dog barks" and "barks the dog" are represented identically.
- **Frequency Count**: Each word's occurrence is counted, and a vector is created where each entry represents the count of a specific word in the vocabulary.

Each sentence is represented as a vector of word counts.

Pro: a simple and efficient way to vectorize text.
Con: lacks the ability to capture i) *word order* or ii) *semantic relationships.*

TF-IDF

An extension of BoW.

TF-IDF (Term Frequency-Inverse Document Frequency):

- TF-IDF assigns a **weight** to each word based on its frequency within a document and across the corpus.
- **Term Frequency (TF)**: Measures the occurrence of a word within a document.
- **Inverse Document Frequency (IDF)**: Reduces the weight of words that are frequent across many documents, *capturing unique terms.*
- This method enhances the BoW model by emphasizing words that are more informative in distinguishing documents.

Word Embedding

- In contrast to BoW and TF-IDF, word embeddings represent words in a **continuous vector space** where **semantically similar** words have similar vectors.
- **Training Process**: Neural networks learn these embeddings in a high-dimensional space, with models like Word2Vec optimizing vectors based on the context in which words appear.
- **Compositionality**: Word embeddings allow for arithmetic operations, showcasing relationships:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

Visualizing Document and Word Embeddings

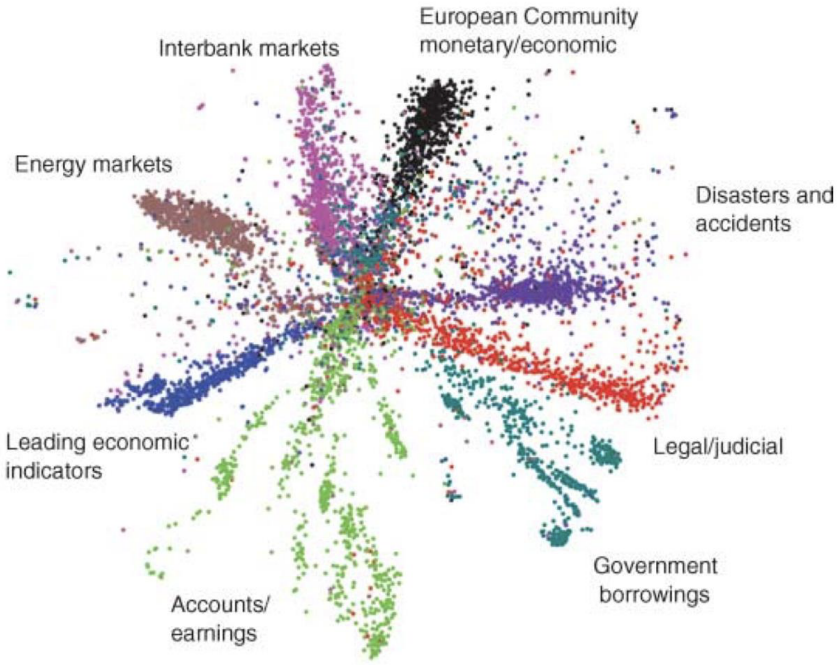


Figure 7.3: **Document embeddings**: where documents are mapped to a space where similar topics are clustered (e.g., *government borrowings*, *energy markets*).

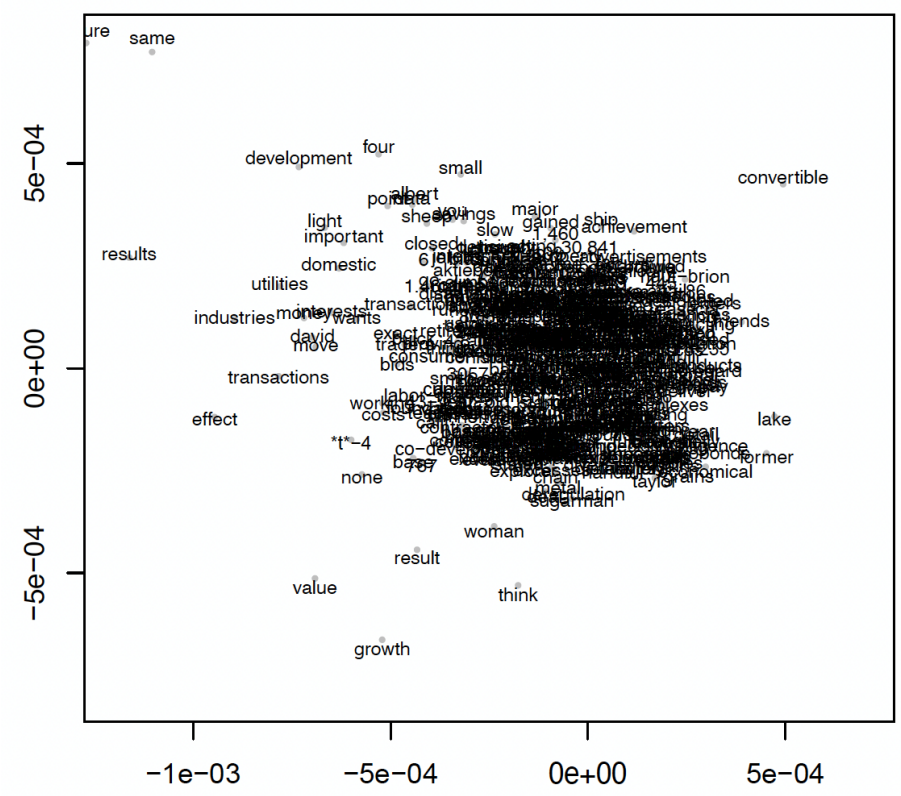


Figure 7.4: **Word embeddings**: as learned by neural networks. Words like "development" and "transactions" are closer due to their contextual similarity, indicating that embeddings capture semantic meanings.

Embeddings reveal structure within text data, organizing information along dimensions that may correspond to latent topics or semantic relationships.

Getting Text Ready for Analysis: NLP Pipelines

Text Preprocessing for NLP

Text preprocessing transforms raw text into a suitable format for model input.

The key steps are:

1. **Loading Text**: Read raw text data into memory as strings.
2. **Tokenization**: Break down text into tokens (e.g., words or sub-words), each representing a meaningful unit for processing.
3. **Vocabulary Creation**: Assign each unique token an index, constructing a dictionary for easy lookup.
4. **Index Conversion**: Convert text into sequences of numerical indices representing tokens.

> NB: this indexing & vocab creation is why when using a pre-trained tokenizer, we need to match the tokenizer to the model.

Additional considerations:

- **Token Granularity**: Tokens may be words, sub-words, or characters, depending on the model's requirements.

- **Special Tokens**: Tokens for rare or unknown words (e.g., "<unk>") are often included to handle out-of-vocabulary cases.

Further Preprocessing Techniques

Beyond tokenization, further steps can improve model performance:

- **Lowercasing**: Converting all text to lowercase for case-insensitive processing.

- **Stop-word Removal**: Removing common but uninformative words like "the" and "and".

- **Stemming**: Reducing words to their base or root form, e.g., *develop, developing, development* become *develop*.

- **Lemmatization**: Mapping words to their dictionary form or lemma, e.g., *drive, drove, driven* becomes *drive*; *easily* becomes *easili*

These preprocessing techniques improve efficiency and accuracy, allowing models to focus on informative content and handle lexical variation effectively.

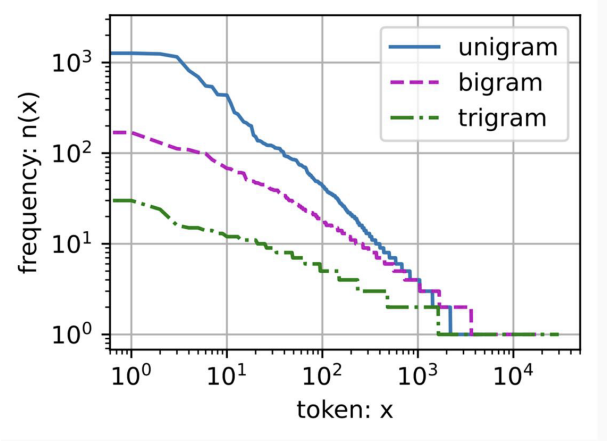


Figure 7.5: Zipf's law: a common characteristic of word frequency distributions, where a small number of tokens occur very frequently, while a large number of tokens occur infrequently; the frequency of tokens decreases as their complexity (unigram, bigram, trigram) increases.

A Simple NLP Pipeline for Document Classification

1. **Tokenization and Preprocessing**: Break down the text into tokens (words or phrases) and apply preprocessing techniques like lowercasing, stop-word removal, etc.
2. **Bag of Words (BoW)**: Represent the document as a vector where each dimension corresponds to the count of a word in the vocabulary, ignoring word order.
3. **TF-IDF Weighting**: Apply *Term Frequency-Inverse Document Frequency* to give more importance to unique terms, thereby enhancing the BoW representation by reducing the influence of common terms.
4. **Classification**: Use the resulting features in a classifier, such as:
   - *Support Vector Machine (SVM)*
   - *Gradient Boosting*
   - *Random Forest*

Advantages:

- Effective for simple tasks, especially when the text's structure or word order is not crucial.
- Results in small, manageable models with fewer parameters to train.

Further Improvements: For more complex tasks or greater accuracy, consider using:

- **Learned Embeddings**: Utilize embeddings that capture word semantics.
- **Advanced Classifiers**: Use classifiers that consider sequence structure, such as LSTMs (Long Short-Term Memory) or Transformers.

General Structure of NLP Using DL

NLP models based on deep learning have a **modular** structure:

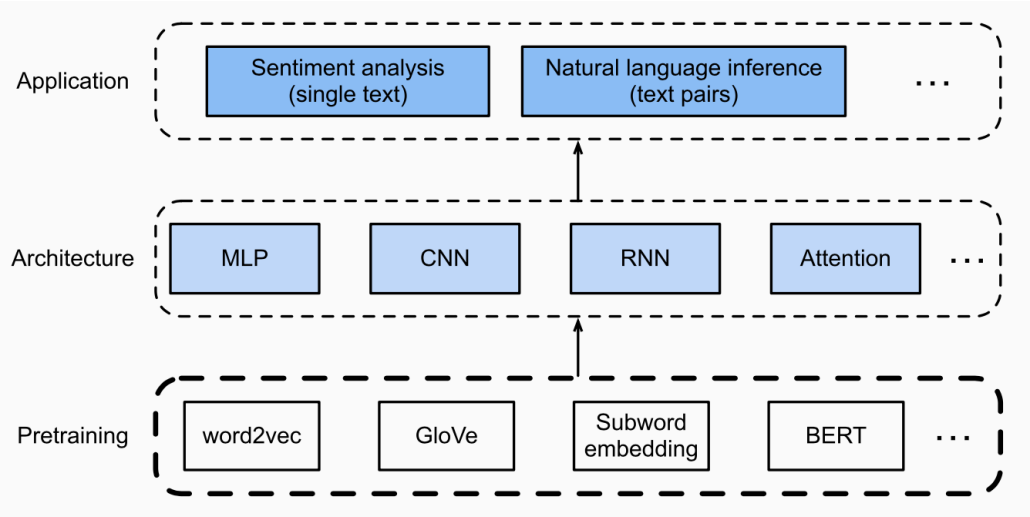


Figure 7.6: NB Bert-Attention are integrated

- **Applications Layer**: Typical NLP tasks include:
  - *Sentiment Analysis*: Classifying the sentiment of a single text, often binary (positive or negative).
  - *Natural Language Inference*: Determining the logical relationship (e.g., entailment or contradiction) between two text segments.
- **Architecture Layer**: The deep learning model architecture varies by task requirements:
  - *MLP (Multi-Layer Perceptron)*: Suitable for simpler tasks that don't require complex context handling.
  - *CNN (Convolutional Neural Network)*: Effective for capturing local patterns, often used in sentence classification.
  - *RNN (Recurrent Neural Network)*: Useful for sequential data as it maintains contextual information across tokens.
  - *Attention Mechanisms*: These allow the model to focus on specific parts of the input text, as in Transformer models.
- **Pretraining Layer**: Pretrained embeddings like *Word2Vec* and *GloVe* provide foundational word vectors, while *BERT* and other contextual models generate embeddings based on surrounding words, capturing nuanced meanings.

Key Points:

- **Modularity**: The process is modular, with embeddings often serving as the base for further feature extraction.
- **Embedding Foundation**: Word embeddings are fundamental to most models and are sometimes integrated directly into them (e.g., BERT).

Embeddings I - One-Hot Encoding

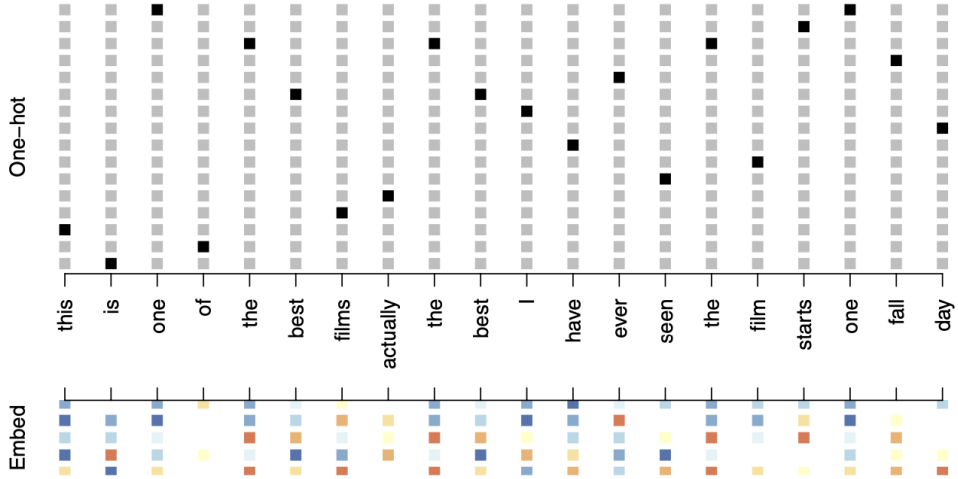


FIGURE 10.13. *Depiction of a sequence of* 20 *words representing a single document: one-hot encoded using a dictionary of* 16 *words (top panel) and embedded in an m-dimensional space with* $m = 5$ *(bottom panel).*

One-hot encoding is a simple method to represent words as vectors in NLP tasks. Here's how it works:

- Each unique word in the vocabulary is represented by a vector of zeros, except for a single position corresponding to that word, which is set to 1.
- For instance, in the sentence "this is one of the best films," each word is assigned a unique vector, such as:

$$\text{“this”} \rightarrow [1, 0, 0, \ldots, 0]$$

- **Vector Length**: The length of each one-hot vector is the **size of the vocabulary**, leading to *high-dimensional* and *sparse* vectors.
- **Limitations**:
  - **No Semantic Similarity**: One-hot vectors are orthogonal and do not capture any semantic relationships between words.
  - **High Dimensionality**: For large vocabularies, one-hot encoding leads to high-dimensional vectors, increasing memory requirements and computation costs.

This lack of semantic relationships between words is addressed by using embeddings, where words are represented in a lower-dimensional, continuous vector space with meaningful relationships.

Embeddings as continuous vector representation, allows calculation of semantic similarity:

$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\|\|B\|}$$

where:

- $A \cdot B$ is the dot product of vectors A and B,
- $\|A\|$ is the magnitude (or norm) of vector A,
- $\|B\|$ is the magnitude (or norm) of vector B.

The dot product $A \cdot B$ can be calculated as:

$$A \cdot B = \sum_{i=1}^{n} A_i B_i$$

And the magnitude of a vector A is given by:

$$\|A\| = \sqrt{\sum_{i=1}^{n} A_i^2}$$

Embeddings II - Word2Vec: Continuous Word Embeddings

A group of **shallow** NNs that learn embeddings.

2 methods: skip-gram & CBOW

The **Word2Vec** model, introduced by Mikolov et al. (2013) at Google, represents words as dense, continuous vectors in a lower-dimensional space.

- **Cosine Similarity**: Used to measure semantic similarity between word vectors, capturing relationships between words based on their contexts.
- **Embedding Dimensions**: Vectors typically have hundreds of dimensions, effectively capturing syntactic and semantic relationships.
- **Training Process**: shallow NNs trained using large corpora in an *unsupervised* manner, learning embeddings *based on surrounding context.*
- **Applications**: Widely used in NLP and extended through models like *doc2vec* for document embeddings and *BioVectors* for biological sequences.
  - doc2vec: used for document retrieval / or for getting specific pages within a larger document - can look for cosing similarity

A notable property of Word2Vec embeddings is their ability to perform arithmetic operations, such as:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

Word2Vec: Skip-Gram and CBOW Models

Skip-Gram and **Continuous Bag of Words (CBOW)** are two architectures in Word2Vec:

- **Skip-Gram Model**: Predicts context words based on a given center word. The objective is to maximize the probability of context words given the center word.

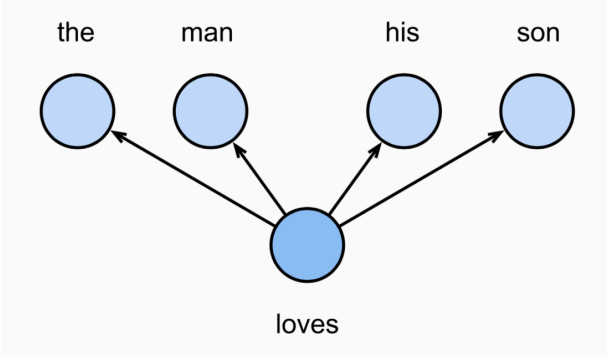


Figure 7.7: Skip-gram model: P("the", "man", "his", "son"|"loves")

- **CBOW Model**: Predicts the center word based on the context words. It seeks to maximize the probability of a word given its surrounding words.

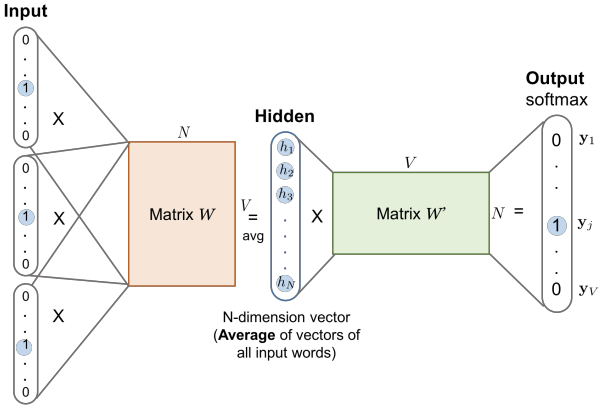


Figure 7.8: CBOW: P ("loves" | "the", "man", "his", "son")

In both models, training involves **adjusting word vectors** to **maximize the likelihood** of context-target pairs, capturing meaningful word relationships.

Skip-Gram Model

Probability and Vector Representation

In the Skip-Gram model, context words are *assumed to be generated independently* given the center word:

$$P(\text{"the", "man", "his", "son"} \mid \text{"loves"}) = P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot \ldots$$

NB: we assume conditional independence given centre word.

This is a basic assumption behind the model.

This means that the probability of observing a set of context words (e.g., "the", "man", "his", "son") given a center word (e.g., "loves") can be decomposed into the product of the individual probabilities of each context word given the center word. The assumption of conditional independence given the center word simplifies the modeling of word relationships.

Each word is represented by two vectors: v_i for the center word and u_i for the context word, both in $\mathbb{R}^d$. I.e. for any word in the dictionary with index i, the vector of it when used as a...

- ... center word is $v_i \in \mathbb{R}^d$.
- ... context word is $u_i \in \mathbb{R}^d$.

NB: with embeddings, we can either choose u or v vector.

Each word appears in both u and v - each word has 2 expressions depending on its context of centre.

Each word in the vocabulary is represented by two vectors of d dimensions: one for when it serves as the center word (v_i) and another for when it serves as a context word (u_i). This dual representation allows the model to capture different aspects of word usage.

Using a softmax operation, we define the conditional probability of a context word w_o given center word w_c:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, n$$

$$P(w_o \mid w_c) = \frac{\exp(u_{w_o}^T v_{w_c})}{\sum_{i \in \mathcal{V}} \exp(u_i^T v_{w_c})}$$

With vocab index set $\mathcal{V} = \{0, 1, \cdots, |\mathcal{V}|, -1\}$.

NB: Use of softmax: A probabilistic model to capture semantic relationships between words.

- Softmax normalizes the score (or "affinity") of each context word relative to the center word, turning **raw similarity scores** into **probabilities**.
- The probabilistic model in the Skip-Gram framework is built around the idea of **maximizing the likelihood of context words given center words**.
- The NN learns to **adjust the vectors (as params)** so that the output (the probabilities of context words) aligns well with the actual observed context words.
- The softmax function ensures that the model outputs a valid probability distribution over all possible context words for a given center word, allowing it to effectively learn from the data in a probabilistic manner.

- During training, the model **optimizes the parameters (word vectors)** so that the predicted probabilities for the observed context words (indicated by the one-hot encoding) are maximized.
- Essentially, the model learns to **adjust the embeddings to increase the predicted probabilities** for actual context words while decreasing the probabilities for incorrect ones.
- The model captures semantic relationships between words based on their context within the training corpus

Explanation of Softmax (with NLP context)

1. **Probability Distribution**: Softmax transforms the dot product $u_{w_o}^T v_{w_c}$ (which measures the similarity between the context and center word vectors) into a probability. This is important because we need $P(w_o \mid w_c)$ to be a valid probability distribution over all possible context words w_o in the vocabulary, meaning all probabilities should sum to 1.

2. **Exponentiation for Emphasis**: Exponentiating the similarity scores (i.e., $\exp(u_{w_o}^T v_{w_c})$) accentuates differences between them, so words with higher similarity to the center word will have a larger impact on the probability. This makes the most similar context words more likely, reflecting the intuition that words in similar contexts should appear together more often.

3. **Computational Efficiency with Log Likelihood**: Softmax allows us to optimize the parameters by maximizing the likelihood of observing actual word pairs in a corpus. The log-likelihood of observed pairs becomes more tractable because it allows us to take derivatives with respect to the parameters u_i and v_i, aiding gradient-based optimization.

4. **Learning Word Embeddings**: Softmax (in combination with the negative sampling or hierarchical softmax techniques often used for large vocabularies) helps the model learn meaningful word vectors u_i and v_i by maximizing the probabilities of observed word pairs. The resulting embeddings capture semantic relationships in the vector space, as the model adjusts vectors to reflect the contexts in which words appear.

Thus, the softmax function serves both as a way to interpret similarity scores as probabilities and as a mechanism for training word embeddings that encode semantic information in a way that aligns with actual word co-occurrences.

Objective Function

The objective of Skip-Gram is to find optimal embeddings u_i and v_i by *maximizing the likelihood of observing context words given the center word*:

For a sequence of length T, with words $w^{(t)}$ at step t, and a context window of size m, the likelihood is:

$$\mathcal{L}(\text{parameters}) = \prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} \mid w^{(t)})$$

Example: "the man loves his son":

- $m = 2$
- $w^{(t)}$ = "loves"

$$\prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} \mid w^{(t)})$$

$$= P(\text{"}the\text{"}_{j=2} | \text{"}loves\text{"}) \cdot P(\text{"}man\text{"}_{j=1} | \text{"}loves\text{"}) \cdot P(\text{"}his\text{"}_{j=1} | \text{"}loves\text{"}) \cdot P(\text{"}son\text{"}_{j=2} | \text{"}loves\text{"})$$

The log-likelihood loss is:

$$-\sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w^{(t+j)} \mid w^{(t)})$$

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log P(\text{"}the\text{"}_{j=2} | \text{"}loves_t\text{"}) + \log P(\text{"}man\text{"}_{j=1} | \text{"}loves_t\text{"}) + \log P(\text{"}his\text{"}_{j=1} | \text{"}loves_t\text{"}) + \log P(\text{"}son\text{"}_{j=2} | \text{"}loves_t\text{"})$$

We would then take the product over all center words in the sequence, not only $w^{(t)} = \text{"}loves\text{"}$.

This formulation encourages embeddings to capture context-based relationships, useful for various NLP applications.

Training

Prediction Task:

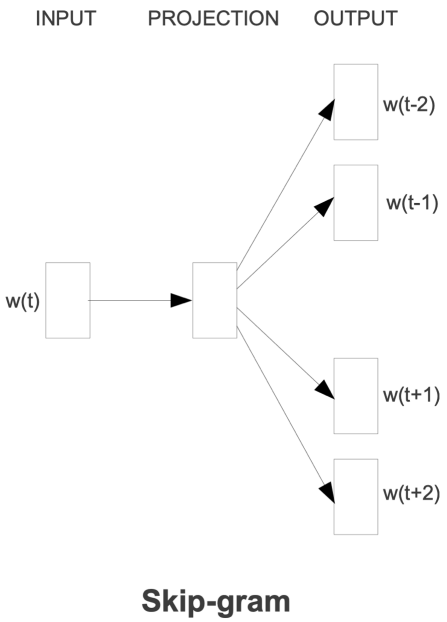


Figure 7.9: Prediction task: predict the context words $w^{(t+j)}$ based on the center word $w^{(t)}$

Prediction task: predict the context words $w^{(t+j)}$ based on the center word $w^{(t)}$.

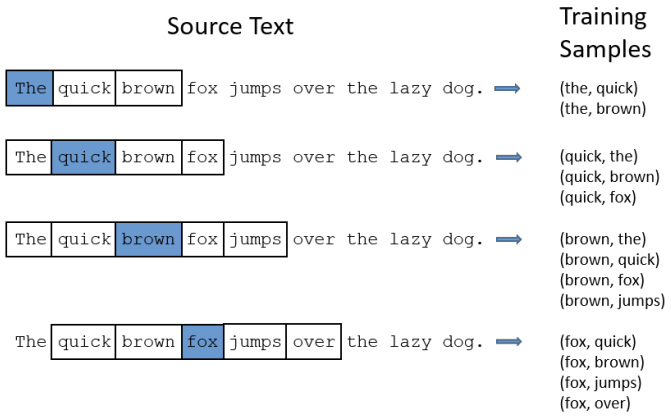
We ultimately **don't care about the prediction**, we want to extract the learned parameters! (the embedding matrix)

In the Word2Vec Skip-Gram model, the model is trained on **word pairs** (of center and context word pairs): training involves **learning word embeddings based on co-occurring word pairs within a specified context window**.

Here is the step-by-step process:

1. **Training Data = Co-occurring Word Pairs**: Each word in a sentence is treated as the center word, and words within its context window are considered context words. Pairs are created from each center word and its surrounding words. For example, in the sentence "The quick brown fox jumps over the lazy dog":

    - With "quick" as the center word, we generate pairs such as ("quick", "the") and ("quick", "brown").



2. **Vocabulary Encoding**: Construct a vocabulary $\mathcal{V}$ and represent each word as a one-hot encoded vector.

3. **Model Input** x: *One-hot encoded* vector representing the **center word**. (Dimension $|\mathcal{V}|$.

4. **Model Output** $\hat{y}$: *Continuous valued* vector representing predicted probabilities for each word in the vocabulary. (Dimension $|\mathcal{V}|$.

5. **Ground Truth** y: *One-hot encoded* vector of the actual context word. (Dimension $|\mathcal{V}|$.

6. **Learned Embeddings**: Through MLE set up from before, the model adjusts its weights so that similar words have embeddings close to each other in the vector space.

This process enables the model to learn word relationships based on their contextual co-occurrence.

Network Architecture

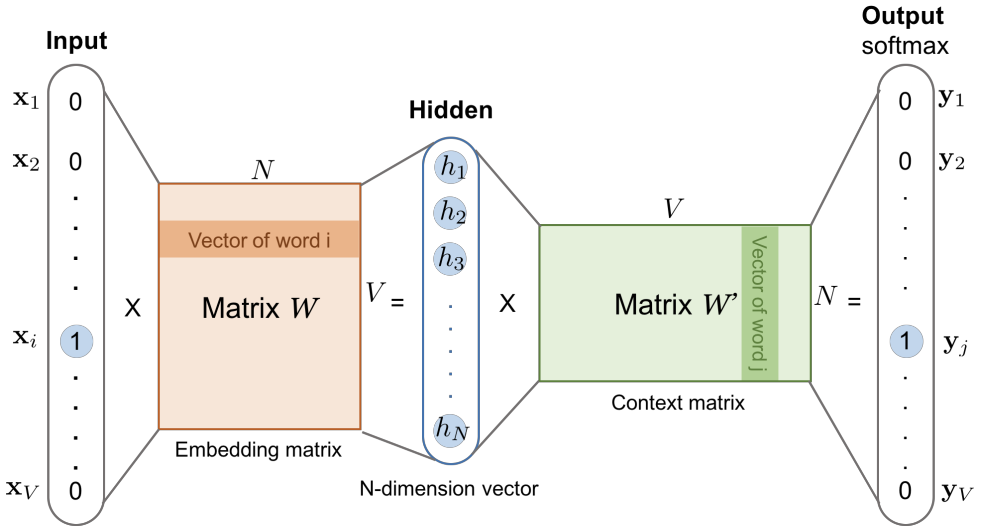


Figure 7.10: Skip-Gram model architecture. Input = centre word v_i; output = context word u_j; matrix vector of interest = v_c which represents centre word (we could also use the context word's representation in matrix vector u_o, but that's more customary for CBOW)

- **Dimensions at each step:**
  1. **Input (One-hot vector of word i)**: $V \times 1$
  2. **After multiplication with W**: $h = W \cdot$ one-hot vector results in $N \times 1$
  3. **After multiplication with W'**: $W' \cdot h$ results in $V \times 1$

- **Input Layer**: A one-hot encoded vector x representing the center word (the target word for which we are predicting context words).
  - **Vector x of word i**: When a specific word i is selected, it is represented as a one-hot vector of dimensions $V \times 1$, where V is the size of the vocabulary (total number of unique words).

- **Embedding Matrix** W: A matrix of dimensions $V \times N$, where:
  - V is the vocabulary size,
  - N is the embedding dimension (size of each word vector).

  This matrix is used to transform the one-hot encoded input x into an embedding v_c for the center word.
  - Since x is a one-hot vector, multiplying it by W effectively selects a single row (embedding) v_c from W, which represents the center word in continuous vector form.
  - **Word embedding** v_c: This vector, of dimensions $N \times 1$, is the continuous-valued embedding of word i and represents the semantic meaning of the center word.

- **Hidden Layer** h: The hidden layer output h is essentially the embedding v_c, obtained by the matrix multiplication $h = W \cdot x$. Therefore, h has dimensions $N \times 1$, representing the embedding of the center word.

- **Context Matrix** W': A matrix of dimensions $N \times V$, where:

  - N is the embedding dimension, matching the dimensionality of h,

  - V is the vocabulary size.

  This matrix serves as an output layer matrix and maps the embedding v_c to a vector of scores for predicting each context word in the vocabulary.

- **Output Layer**: The hidden layer h (which is v_c) is multiplied by the context matrix W' to produce a score vector $z = W' \cdot h$ of dimensions $V \times 1$. Each entry in z corresponds to a score for a word in the vocabulary, indicating the likelihood of each word being a context word for the center word.

- **Softmax Layer**: The final output $\hat{y}$ is obtained by applying the softmax function to the score vector z, generating a probability distribution over all words in the vocabulary. Each element $\hat{y}_j$ represents the probability that word j is a context word given the center word i.

In the Skip-gram and Continuous Bag of Words (CBOW) models, such as the one represented in the diagram, there is no non-linear activation between the embedding and output layers. Instead, the architecture relies purely on linear transformations followed by a softmax function in the output layer to learn useful word embeddings.

1. **Embedding Interpretation**:
    - When we multiply the one-hot vector x by the embedding matrix W, we're effectively selecting a single row (vector) from W, corresponding to the embedding v_c of the center word. So, in this context, h **is just the embedding vector** v_c **from within** W**.**
    - This vector v_c (dimension $N \times 1$) acts as the learned representation of the word, capturing its semantic properties based on co-occurrence with context words.
2. **Why No Activation Doesn't Lead to Collapse**:
    - The model doesn't collapse because the training objective is to maximize the likelihood of predicting correct context words around the center word. The model uses a softmax layer at the output to adjust the word embeddings so that W and W' together learn meaningful relationships between words.
    - The softmax function in the output layer, combined with backpropagation, encourages the embeddings in W to spread out in the N-dimensional space in a way that reflects semantic similarity. Words that appear in similar contexts will have similar embeddings, but without collapsing to the same values, as that would prevent the model from distinguishing them.
3. **Role of** W **and** W':
    - The two matrices, W (input embedding matrix) and W' (output/context embedding matrix), work together in training. During backpropagation, they are updated independently, which allows the model to learn nuanced distinctions between words based on context.
    - While W is used to generate the word embedding, W' transforms this embedding into a space where the model can compute probabilities over the vocabulary. The separation of W and W' further prevents the model from collapsing, as the output scores are derived from a different transformation than the embedding itself.
4. **Effect of the Loss Function**:
    - The negative log-likelihood loss (or cross-entropy loss) encourages the model to adjust W and W' so that context words have high probabilities near each center word and low probabilities otherwise. This gradient-based optimization indirectly promotes diversity among the embeddings in W, which prevents collapse.

In summary, h is just the vector v_c from within W, representing the embedding of the center word. The absence of a non-linear activation function doesn't lead to collapse because the training objective, through the softmax and cross-entropy loss, implicitly regularizes the embeddings. This setup allows the model to learn rich and distinct word embeddings that capture semantic relationships based on contextual co-occurrences.

Loss Function and Gradient Update

Loss Function:

The Skip-Gram model aims to maximize the likelihood of observing context words w_o (output words) given a center word w_c.

The Skip-Gram model uses the log-likelihood of observed word pairs as its loss function. For a word pair (w_o, w_c), the loss is given by:

$$\begin{aligned}\log P(w_o \mid w_c) &= \log \frac{\exp(u_o^T v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^T v_c)} \\ &\textit{...apply some log rules...} \\ &= u_o^T v_c - \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^T v_c) \right)\end{aligned}$$

where u_o and v_c are the vectors for the context and center words, respectively

Now we want to take derivative wrt to weights we want to learn (essentially back propagation).

Deriving the log-likelihood loss function

1. **Conditional Probability of Context Word**: The probability of observing a context word w_o given a center word w_c is modeled using the softmax function. For each word pair (w_o, w_c), we define this probability as:

$$P(w_o \mid w_c) = \frac{\exp(u_o^T v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^T v_c)}$$

where:

- u_o is the output vector for the word w_o,
- v_c is the input (center) vector for the word w_c,
- $\mathcal{V}$ represents the vocabulary set (all possible words).

2. **Log-Likelihood of Observed Pair**: The Skip-Gram model maximizes the log-likelihood of observed word pairs (w_o, w_c). Thus, for a given word pair, the log-likelihood is:

$$\log P(w_o \mid w_c) = \log \left(\frac{\exp(u_o^T v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^T v_c)} \right)$$

3. **Simplifying the Log-Likelihood Expression**: Using the properties of logarithms, we can separate this expression:

$$\log P(w_o \mid w_c) = \log \left(\exp(u_o^T v_c) \right) - \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^T v_c) \right)$$

$$= u_o^T v_c - \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^T v_c) \right)$$

4. **Final Loss Function**: Therefore, for a single word pair (w_o, w_c), the Skip-Gram model's loss function, which we seek to minimize, is:

$$-\log P(w_o \mid w_c) = -\left(u_o^T v_c - \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^T v_c) \right) \right)$$

This expression captures the log-likelihood of observing a particular context word w_o given a center word w_c.

By maximizing this log-likelihood over all observed word pairs in the corpus, the Skip-Gram model learns embeddings u and v for each word that capture semantic relationships based on word co-occurrences.

we can compute a gradient update with respect to the parameters (e.g. the center word vector v_c (as v_c *is* a vector of learned parameters), by taking the derivative of the loss.

Gradient Update: To update v_c (center word embedding), we calculate the gradient of the loss with respect to v_c:

$$\frac{\partial \log P(w_o \mid w_c)}{\partial v_c} = u_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c) u_j$$

Where:

- $\frac{\partial \log P(w_o|w_c)}{\partial v_c}$: The gradient of the log-probability of observing the context word w_o given the center word w_c with respect to the center word vector v_c.
- u_o: The vector representation of the observed context word w_o.
- $P(w_j \mid w_c)$: The probability of observing the word w_j given the center word w_c, calculated using the softmax function.
- u_j: The vector representation of the context word w_j.
- $\mathcal{V}$: The vocabulary set, which includes all words in the model.

Intuition: these updates help optimize the embeddings to maximize the likelihood of *context words*.

$$\begin{aligned}\frac{\partial \log P(w_o \mid w_c)}{\partial v_c} &= u_o - \frac{1}{\sum_{i\in\mathcal{V}} \exp(u_i^T v_c)} \sum_{j\in\mathcal{V}} \exp(u_j^T v_c) u_j \\ &= u_o - \sum_{j\in\mathcal{V}} \left(\frac{\exp(u_j^T v_c)}{\sum_{i\in\mathcal{V}} \exp(u_i^T v_c)} \right) u_j \\ &= u_o - \sum_{j\in\mathcal{V}} P(w_j \mid w_c) u_j\end{aligned}$$

Explanation:

1. **Objective**: We are computing the gradient of $\log P(w_o \mid w_c)$ with respect to the center word vector v_c. Formally, we want to optimise:

$$\frac{\partial \log P(w_o \mid w_c)}{\partial v_c}$$

This gradient will update the word vectors v_c for the center word (and u_o for the context words.)

2. **Log-Probability**: Recall the log-probability expression for observing the context word w_o given the center word w_c:

$$\log P(w_o \mid w_c) = u_o^T v_c - \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^T v_c) \right)$$

3. **Differentiating with Respect to** v_c: To find the gradient $\frac{\partial \log P(w_o|w_c)}{\partial v_c}$, we take the partial derivative of the log-probability. Applying the chain rule, we get two terms:

$$\frac{\partial \log P(w_o \mid w_c)}{\partial v_c} = \frac{\partial}{\partial v_c}(u_o^T v_c) - \frac{\partial}{\partial v_c} \log \left(\sum_{i \in \mathcal{V}} \exp(u_i^T v_c) \right)$$

- The first term, $\frac{\partial}{\partial v_c}(u_o^T v_c)$, simplifies to u_o.
- The second term involves the derivative of the log-sum-exp function, which leads to:

$$-\frac{1}{\sum_{i \in \mathcal{V}} \exp(u_i^T v_c)} \sum_{j \in \mathcal{V}} \exp(u_j^T v_c) u_j$$

4. **Rewriting in Terms of Probabilities**: The expression can be simplified by recognizing that $\frac{\exp(u_j^T v_c)}{\sum_{i \in \mathcal{V}} \exp(u_i^T v_c)}$ is the softmax probability $P(w_j \mid w_c)$ of the word w_j given the center word w_c. This gives:

$$= u_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c) u_j$$

***Interpretation**: The gradient $\frac{\partial \log P(w_o|w_c)}{\partial v_c}$ shows the difference between the vector for the observed context word u_o and a weighted average of all context word vectors u_j in the vocabulary, weighted by their probabilities $P(w_j \mid w_c)$.*

This gradient drives the model to adjust v_c such that u_o becomes more similar to the predicted distribution of context words, thereby capturing semantic relationships.

Negative Sampling

Problem with Computing Embeddings:

Our final objective function to maximize is:

$$\mathcal{L} = u_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c) u_j$$

The problem is that $\mathcal{V}$ is the vocabulary; this means we need to compute the conditional probabilities of *all* the words in the vocabulary conditional on the center word.

This is a *huge* task given that the vocabulary can have millions of tokens.

The trick is to add some noise through **negative sampling** to approximate the loss function.

Negative Sampling is a technique used to reduce the computational costs in training word embeddings by *approximating the loss function* through selective sampling of negative (noise) data points.

- **Positive Data Point**: These are context word pairs that occur naturally within a defined context window in the corpus. For example, the words ("brown", "quick") might appear within the same window, indicating that they co-occur and should therefore reinforce each other in the model.
- **Negative (Noise) Data Point**: These are randomly selected word pairs that do not appear together within the context window. For example, the pair ("fox", "dog") might be selected as a noise example if they do not co-occur within a specific context window. Negative sampling uses such noise data points to differentiate true context pairs from random, unrelated word pairs.
- **Objective**: For each context pair, the objective is to:
  - Maximize the probability $P(D = 1 \mid w_c, w_o)$ if the pair (w_c, w_o) is a true context pair (observed within the context window).
  - Maximize the probability $P(D = 0 \mid w_c, w_o)$ if the pair (w_c, w_o) is a randomly generated noise pair.

  This is **achieved using a sigmoid function** to calculate the probabilities:

  $$P(D = 1 \mid w_c, w_o) = \sigma(u_o^T v_c)$$

  where σ represents the sigmoid function.

  We *wrap* the original dot product expression in a sigmoid function to model these new conditional probabilities, effectively distinguishing between positive and negative samples.

  This approach allows us to approximate the likelihood function (and hence the loss function) by using a set of K negative samples rather than computing over all possible word pairs in the vocabulary.
- **Efficiency**: Negative sampling significantly reduces the computational cost because the model only needs to compute gradients for a small number K of noise samples, rather than

for all possible pairs in the vocabulary. This leads to computational costs that *scale linearly* with the number of negative examples K rather than with the size of the vocabulary.

In summary, negative sampling allows for efficient training by focusing on a contrastive objective that learns to distinguish true context pairs from noise pairs, effectively balancing training accuracy and computational efficiency.

Continuous Bag of Words (CBOW) Model

CBOW is an alternative to the Skip-Gram model where the objective is to predict the center word from surrounding context words. Key characteristics include:

- **Objective**: Given context words $(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2})$, predict the center word w_t.
- **Architecture**: Similar structure to Skip-Gram but uses an *average* of the context word embeddings as input to predict the center word.
- **Word embeddings are typically the context word vectors**. (Rather than the centre words for Skip-Gram models)
- **Use Case**: CBOW is more suitable for *smaller datasets* as averaging context embeddings can *reduce overfitting*.

CBOW's averaging of context word vectors helps reduce noise, making it more robust in limited data scenarios.

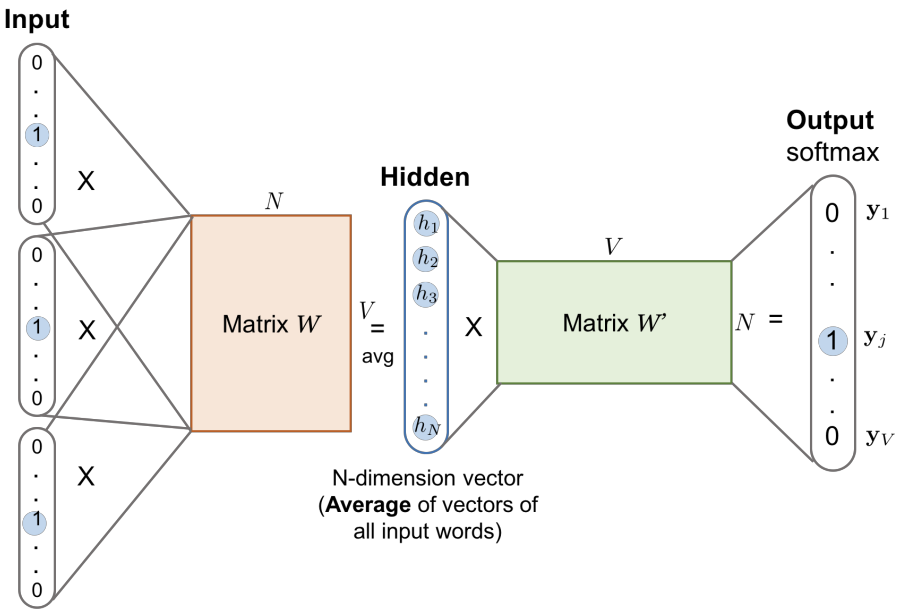


Figure 7.11: In the CBOW architecture, the embeddings for the context words are indeed represented by the weight matrix W'

CBOW Model Architecture and Dimensions

This figure represents the architecture of the Continuous Bag of Words (CBOW) model, where multiple context word vectors are averaged to predict a target word. In this model, two weight matrices, W and W', are used to transform input and output vectors, and the word embeddings are typically derived from the context word vectors in W.

- **Input Layer**:
  - The input consists of multiple one-hot encoded vectors representing context words surrounding the target word.
  - Each one-hot vector has a dimension of V, where V is the vocabulary size.
  - The number of context words (inputs) is denoted by C.
- **Embedding Layer (Matrix W)**:
  - Matrix W is a weight matrix of dimensions $V \times N$, where N is the embedding size.
  - For each context word, the one-hot encoded input vector selects the corresponding row in W, producing an embedding vector of dimension N.
  - The embeddings for each context word are then averaged to produce the hidden layer vector h, which also has dimension N.
  - This averaging step captures the overall semantic context around the target word.
  - The rows of W corresponding to each word in the vocabulary serve as the context word embeddings in the CBOW model.
- **Hidden Layer Representation h**:
  - h is an averaged dense vector of dimensions N, representing the combined semantic information from all context words.
  - This vector h is then multiplied by the output weight matrix W' to predict the target word.
- **Output Layer (Matrix W')**:
  - Matrix W' is a weight matrix of dimensions $N \times V$.
  - When the hidden layer vector h is multiplied by W', it produces a vector of dimension V, which corresponds to the scores (logits) for each word in the vocabulary.
  - These scores are passed through a softmax function to produce a probability distribution over the vocabulary, representing the likelihood of each word being the target word given the context words.
- **Final Output**:
  - The output is a probability distribution of dimension V, where each element represents the predicted probability of a word in the vocabulary being the target word given the context words.

Summary of Dimensions:

Input vectors (one-hot encoded for each context word):	$V \times 1$ (each of C inputs)
Weight matrix W:	$V \times N$
Hidden layer vector h (averaged context embeddings):	$N \times 1$
Weight matrix W':	$N \times V$
Output vector (logits):	$V \times 1$

In the CBOW model, the context word embeddings are learned in matrix W, and these embeddings capture the semantic relationships of words based on the surrounding context they appear in.

Embeddings III: Word Embedding with Global Vectors (GloVe)

GloVe (Global Vectors for Word Representation) is a method for learning word embeddings by using global word co-occurrence statistics from a corpus. It differs from the Skip-Gram and Continuous Bag of Words (CBOW) models in the following ways:

- **Modification of Skip-Gram Model**: GloVe can be seen as an extension or modification of the Skip-Gram model but emphasizes capturing global co-occurrence statistics across the entire corpus.
- **Symmetric Co-occurrences**: Unlike Skip-Gram, which models asymmetric conditional probabilities (e.g., $P(\text{context} \mid \text{center word})$), GloVe relies on symmetric co-occurrence counts, capturing how frequently pairs of words appear together.
- **Center and Context Equivalence**: In GloVe, the embedding of the center word and context word are mathematically equivalent for any word. This symmetry is achieved by modeling their interaction directly.
- **Squared Loss Function**: Instead of using a log-likelihood, GloVe uses a squared loss function to fit the embeddings based on precomputed global statistics. This allows it to capture more meaningful relationships between words across the corpus.

GloVe is designed to combine the advantages of both local context window methods (like Skip-Gram) and global matrix factorization methods (like Latent Semantic Analysis) by using global co-occurrence statistics to build embeddings.

Embeddings IV: Contextual Word Embeddings

Contextual word embeddings represent each word differently based on its surrounding context, addressing the limitation of traditional embeddings like Word2Vec and GloVe that assign a single embedding to each word regardless of its usage.

- **Word Sense Disambiguation**: In static embeddings, a word like "bank" has the same embedding whether it refers to a financial institution or the side of a river. Contextual embeddings adjust based on usage, as shown in sentences:
  - "I have money in the **bank**."
  - "Bank fishing a river is a great way to catch a lot of smallmouth bass."
- **Handling Polysemy**: Many words have multiple meanings. Contextual embeddings dynamically adjust the vector for a word to reflect its specific meaning in the given sentence or paragraph.
- **Deep Learning Approaches**: Modern NLP models, such as BERT (Bidirectional Encoder Representations from Transformers), rely on deep learning to produce embeddings that change according to context, providing a more accurate representation of language.
- **Benefits**: Contextual embeddings offer a more nuanced understanding of language, as they capture the specific meaning of a word in each context, which is essential for tasks like sentiment analysis, machine translation, and question answering.

Sentiment Analysis

Sentiment Analysis with Recurrent Neural Networks (RNN)

Sentiment analysis is the task of classifying a piece of text (e.g., a sentence, tweet, or review) as expressing a particular sentiment, such as *positive*, *negative*, or *neutral*.

This process is widely used in applications where understanding users' opinions is important, like customer feedback analysis or social media monitoring.

In this context, RNNs (Recurrent Neural Networks) play a critical role due to their ability to handle sequential data and capture dependencies over time. (Varying length text sequence will be transformed into fixed categories)

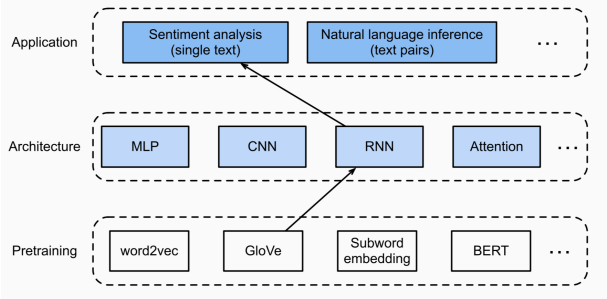


Figure 7.12: Sentiment analysis approach: combines a pretraining approach with DL architecture to perform the classification task.

Basic RNNs for Sentiment Analysis

RNNs are specifically designed to handle sequential data. Unlike traditional feedforward neural networks, an RNN processes each word in the sequence one at a time, maintaining a hidden state that captures information about previous words in the sequence.

- **Input Layer**: At each time step t, the input x_t represents the word embedding of the current word in the sequence.
- **Hidden Layer**: The hidden state h_t at time t depends on the input x_t and the previous hidden state h_{t-1}. This dependency allows the RNN to capture sequential patterns.
- **Output Layer**: At each time step, the output o_t can represent the predicted sentiment, and the final output can be taken after processing all time steps in the sequence.

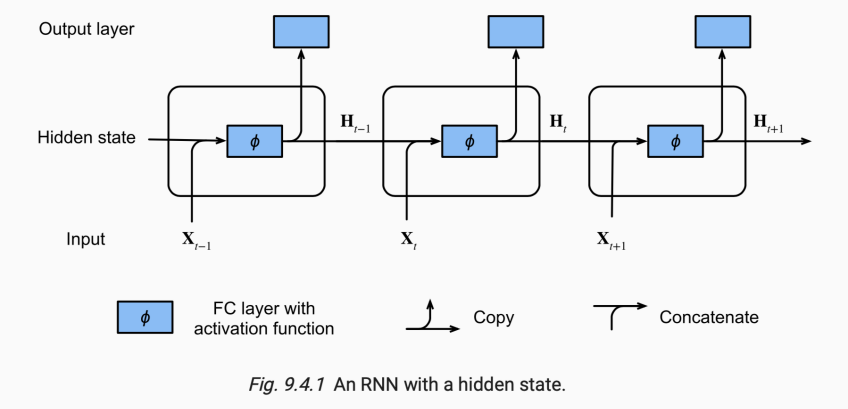


Fig. 9.4.1 An RNN with a hidden state.

The hidden state h_t at each time step is updated according to:

$$h_t = f(W_h x_t + U_h h_{t-1} + b_h)$$

where W_h and U_h are weight matrices, b_h is a bias term, and f is an activation function (often tanh).

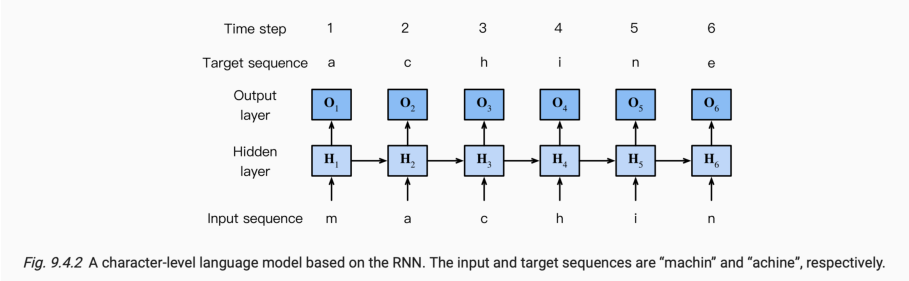


Fig. 9.4.2 A character-level language model based on the RNN. The input and target sequences are "machin" and "achine", respectively.

Figure 7.13: Character sequence modeling: [m,ac,h,i,n.e]

Challenges with Basic RNNs

RNNs can capture dependencies in a sequence, but they struggle with long-range dependencies due to issues like vanishing gradients. This limitation affects tasks that require understanding the sentiment of a text, especially when sentiment is determined by words far apart in the sequence.

Exponential Forgetting:

RNNs: Recurrence leads to exponential forgetting wrt the distance of time steps. Past information gets progressively "forgotten" as time moves forward. This is primarily due to the nature of the sigmoid activation function and the recurrent connections. The recurrence causes earlier states to contribute less to the current output as time passes, with the effect of each past state diminishing exponentially. This means that long-term dependencies are harder for RNNs to capture effectively.

LSTMs which are designed to mitigate the vanishing gradient problem, the recurrence still exists, so they too suffer from exponential forgetting to some extent. While LSTMs are better at preserving long-term information due to their gating mechanisms (like the forget and input gates), they can still forget information exponentially over time, especially when the time window is large and the gating mechanism doesn't sufficiently retain relevant information.

Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU)

LSTM and GRU networks were introduced to address the limitations of standard RNNs. Both architectures include mechanisms (gates) to control the flow of information through the network, allowing them to capture long-term dependencies more effectively.

- **LSTM**: Includes input, forget, and output gates to decide what information to keep, forget, or output. This helps in retaining relevant information over long sequences.
- **GRU**: A simpler variant of LSTM, combining the forget and input gates into a single update gate. GRUs are computationally efficient and perform well on tasks like sentiment analysis.

Bidirectional RNNs

Bidirectional Recurrent Neural Networks (Bidirectional RNNs) are an extension of standard RNNs that capture context from both directions in a sequence.

While a regular RNN processes information from the beginning to the end of the sequence (left to right), a bidirectional RNN consists of two RNNs: one that processes the sequence from start to end (forward) and another that processes it from end to start (backward).

This structure allows the model to **utilize information from both past and future words for each word in the sequence**, which is particularly beneficial for tasks like sentiment analysis, where understanding the full context of a sentence is critical.

For example, in a sentence like "I am not happy," the word "not" changes the sentiment, and its context needs to be captured both from preceding and succeeding words. By using both forward and backward contexts, bidirectional RNNs can better understand such dependencies within a sentence.

Bidirectional RNN Architecture

In a bidirectional RNN, the hidden state H_t at each time step t is a concatenation of the forward hidden state $\overrightarrow{H_t}$ and the backward hidden state $\overleftarrow{H_t}$:

$$H_t = \overrightarrow{H_t} \oplus \overleftarrow{H_t}$$

where $\oplus$ represents concatenation. As such,

$$H_t \in \mathbb{R}^{n \times 2h}$$

Where h is the number of hidden units in each direction.

H_t is the fed into the output layer.

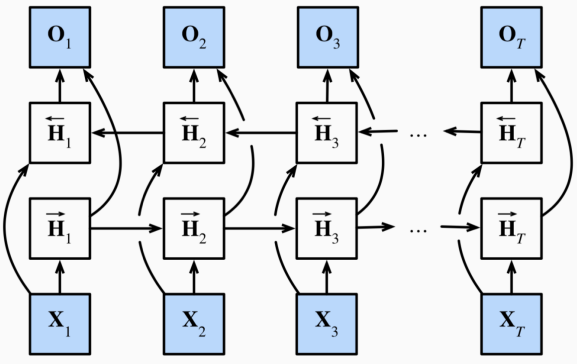


Fig. 10.4.1 Architecture of a bidirectional RNN.

This combined hidden state provides richer contextual information, as it considers both previous and upcoming words relative to t.

Pretraining Task: Masked Language Modeling

A common pretraining task for bidirectional models, especially in the context of language models like BERT, is **masked language modeling**. In this task, specific tokens in a sentence are masked (hidden), and the model is trained to predict the masked tokens based on the surrounding context. This process helps the model learn to fill in missing information by leveraging both the preceding and following words.

For example, consider the following table where the goal is to predict a masked word based on the sentence context:

Sentence	**Options**	**Removed**
I am _____.	happy, thirsty	-
***Comment**: can be basically anything*		
I am _____ hungry.	very, not	happy, thirsty
***Comment**: now needs to be an adverb*		
I am _____ hungry, and I can eat half a pig.	very, so	not
***Comment**: now quite specific*		

Table 7.1: Intuition - what comes later downstream is informative of the previous word.

In this task:

- For the sentence "I am ___," both "happy" and "thirsty" could fit reasonably well, given no additional context.
- In the sentence "I am ___ hungry," options are narrowed down as "very" or "not" would make more sense than "happy" or "thirsty" in this context.
- Similarly, in the sentence "I am ___ hungry, and I can eat half a pig," the words "very" or "so" are likely choices, but "not" would be inappropriate here.

Bidirectional! Very different from forecasting: in text you do want what is in the future to help train your model.

This masked language modeling pretraining task teaches the model to capture nuanced dependencies and context for each word position, helping it perform better in downstream tasks such as sentiment analysis, where understanding the entire context is essential.

Training with Sentiment Labels

During training, the model is provided with text sequences and corresponding sentiment labels. The goal is to minimize the loss between the predicted sentiment and the actual label. A common loss function is cross-entropy loss for multi-class classification tasks:

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

where y_i is the true label, $\hat{y}_i$ is the predicted probability for class i, and N is the number of classes (e.g., positive, negative, neutral).

Example: Sentiment Analysis on Movie Reviews

Consider a dataset of movie reviews labeled as positive or negative. The model processes each review as a sequence of word embeddings, capturing the sentiment context through RNN layers. Over training epochs, the model learns to associate certain words or patterns with positive or negative sentiment.

Example Architecture for Sentiment Analysis:

- Input Layer: Word embeddings of each word in a review.
- RNN Layer: Processes the sequence to capture temporal dependencies.
- Fully Connected Layer: Maps the RNN output to sentiment classes.
- Output Layer: Softmax activation to predict the probability of each sentiment class.

This architecture can be further enhanced by using pre-trained embeddings like `word2vec`, `GloVe`, or contextual embeddings from `BERT` for richer semantic understanding of words.

Regularization in Deep Learning

Regularization is a set of techniques used to prevent overfitting in machine learning models. Overfitting occurs when a model performs very well on the training data but poorly on unseen validation data, indicating that the model has learned noise rather than underlying patterns.

Can also help to make models computationally efficient

Types of Regularization Techniques

1. Weight Sharing

Weight sharing reduces the number of parameters in a model by enforcing parameter reuse, which helps in preventing overfitting and makes models computationally efficient. It is commonly applied in two contexts:

- **Convolutional Neural Networks (CNNs):** In CNNs, the same filter (set of weights) is applied to different parts of the input image, thus learning spatial hierarchies and reducing the number of parameters.
- **Recurrent Neural Networks (RNNs):** In RNNs, the weights are shared across each time step. For an RNN with hidden state H_t, we have:

  $$H_t = g(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$$

  where W_{xh} is the input-to-hidden weight, W_{hh} is the hidden-to-hidden weight, and b_h is the bias term. These weights are shared across time steps, which helps capture temporal dependencies without increasing model complexity.

2. Weight Decay (L2 Regularization)

Weight decay, also known as L_2 regularization, adds a penalty term to the loss function that penalizes large weights, thus encouraging the model to learn simpler patterns. This approach is inspired by regularization techniques in linear models like LASSO and ridge regression.

The new objective function becomes:

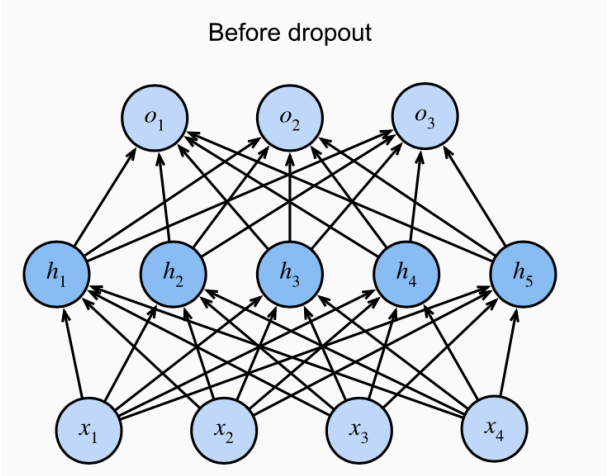
$$L_{\text{new}} = L_{\text{original}}(W) + \lambda \|W\|_2^2$$

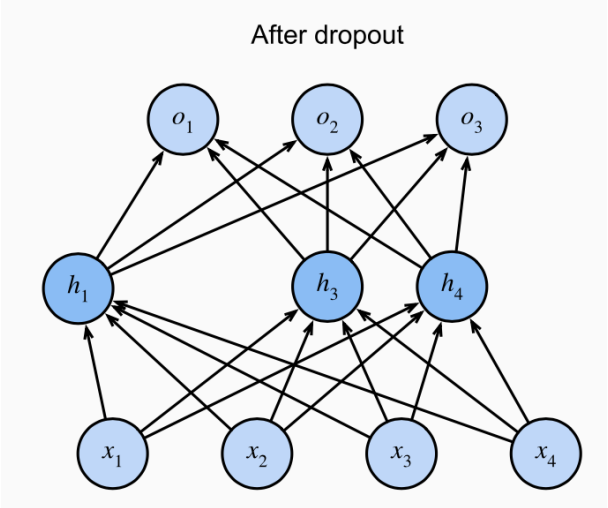
where λ is a regularization parameter controlling the trade-off between minimizing the original loss **and** minimizing the magnitude of the weights.

In gradient descent, this penalty leads to a "decay" in the weight update, effectively shrinking weights towards zero over time, thereby reducing model complexity.

3. Dropout

Dropout is a *stochastic regularization* technique that involves randomly "dropping out" (setting to zero) a subset of neurons during each training iteration.
This *introduces noise* into the model, forcing it to learn robust features rather than overfitting to specific paths in the network.





The procedure involves:

- At each training iteration, randomly zero out a fraction of nodes in each layer.
- During forward propagation, only a subset of the neurons are active, which effectively creates an ensemble of different network configurations.

- During inference (testing), dropout is turned off, and the full network is used by scaling the activations to maintain the expected output.

This process prevents co-adaptations of neurons and thus reduces overfitting, as the model cannot rely on any single pathway for making predictions.

Benefits of Regularization

Regularization techniques help:

- Improve generalization by reducing model complexity.
- Prevent overfitting, where training performance is high, but validation performance is poor.
- **Enhance computational efficiency, especially through weight sharing**, by reducing the number of parameters that need to be stored and computed.