Deep Learning Course Notes

Notes by Colin Young

https://yuhao-yang-cy.github.io/

2026-04-25

> These are a set of notes that I took as I worked my way through the Deep Learning Specialisation, a fabulous series of courses on AI technology on Coursera taught by Andrew Ng.
>
> Almost all contents in the notes, including the texts, formulas and figures are either taken from the course materials or from internet resources. I have also received much help from Claude for revising my notes.
>
> All credits of these notes and my deepest respect go to Andrew Ng and his teaching team.

Deep Learning Course Notes

- Course 1A: Binary Classifiers with Logistic Regression
  - Forward Propagation
  - Estimation of Loss
  - Vectorisation
  - Activation Functions
    - Sigmoid
    - Hyperbolic Tangent
    - ReLU (Rectified Linear Unit)
    - Leaky ReLU
    - How to Choose Activation Functions?
  - Gradient Descent and Backward Propagation
    - Backward Propagation for Cross-entropy
  - Implementation of Binary Classifier
- Course 1B: Multi-layer Neural Networks
  - Deep L-layer Neural Network
    - Notations
  - Forward Propagation
  - Backward Propagation
    - Derivative for Neurons
    - Derivative for Weights
    - Derivative for Biases
    - Derivative for Activations in the Previous Layer
    - Backward Propagation Functions Wrap-up
  - Vectorisation with m Training Examples
  - Typical Architecture of a Deep Neural Network
- Course 2: Hyper-parameter Tuning, Regularization and Optimization
  - Bias-Variance Trade-off
    - Derivation of the Bias–Variance Decomposition
    - Recipe for Machine Learning
  - Regularisation
    - L_2-regularisation
    - L_1-regularisation
    - Other Regularisation Methods
  - Initialisation
    - Normalising Inputs
    - Weight Initialisation
  - Optimisation Algorithms
    - Mini-batch Gradient Descent
    - Gradient Descent with Momentum
    - RMSProp (Root Mean Square Propagation)
    - Adam Optimisation Algorithm
  - Hyperparameter Tuning
    - Examples of Hyperparameters
    - Batch Normalisation
  - Multi-class Classification
    - SoftMax Activation
    - Backward Propagation for $dZ^{[l]}$

- SoftMax with Temperature
- Course 3-1: Convolutional Neural Network Basics
  - Convolutional Filtering
    - Convolutional Filters
    - Why Convolutional Operation?
    - Padding
    - Strided Convolutions
    - Convolution over Volumes
    - Summary of Convolutional Filtering (One Layer)
  - Pooling
    - Max Pooling
    - Average Pooling
  - Convolutional Neural Networks
    - Typical CNN Architectures
  - Backward Propagation for CNN
    - Backward Pass for Pooling Layers
    - Backward Pass for Convolutional Layers
  - Implementation Notes
    - Forward Propagation
    - Backward Propagation
- Course 3-2: CNN Architecture Case Studies
  - Residual Networks (ResNets)
    - Why ResNets Work?
    - Typical Residual Blocks
  - Depthwise Separable Convolutions
    - MobileNetV2 Architecture
  - Inception Networks
  - Further Advices
    - Transfer Learning
    - Data Augmentation
- Course 3-3A: Object Detection
  - Typical Tasks in Computer Vision
  - Object Detection
    - Defining the Outputs
    - Landmarks
    - Choosing the Loss Function
    - Sliding Window Technique
    - Sliding Window with Convolutional Implementation
  - YOLO
    - Bounding Box Prediction
    - Anchor Boxes
    - Class Scores
    - Non-Max Suppression
    - Visualising YOLO
- Course 3-3B: Semantic Segmentation
  - Semantic Segmentation: Overview
  - Fully Convolutional Networks (FCNs)
    - Transpose Convolution
  - U-Net
    - U-Net: Model Details
    - Experimental Results
- Course 3-4: Face Recognition & Neural Style Transfer
  - Face Recognition
    - One-shot Learning
    - Siamese Network
    - Triplet Loss
    - Binary Classification
  - Neural Style Transfer
    - Choosing the Layers of the Model
    - Cost Functions for NST
    - Content Cost Function $J_{\text{content}}(C, G)$
    - Style Matrix
    - Style Cost
    - Total Cost

- Course 4-1A: Recurrent Neural Networks
  - Recurrent Neural Networks
    - Why Not Standard Neural Networks?
    - Features of Recurrent Neural Networks
    - Types of RNNs
    - Problems with RNNs
  - Basic RNN Cells
    - Forward Propagation
    - Backward Propagation
- Course 4-1B: LSTM & GRU Networks
  - Long Short-Term Memory (LSTM) Networks: Forward Propagation
    - Forget gate $\mathbf{\Gamma}_f$
    - Candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$
    - Update gate $\mathbf{\Gamma}_u$
    - Cell state $\mathbf{c}^{\langle t \rangle}$
    - Output gate $\mathbf{\Gamma}_o$
    - Prediction $\mathbf{y}^{\langle t \rangle}_{\text{pred}}$
    - Summary
  - Long Short-Term Memory (LSTM) Networks: Backward Propagation
    - Gate Derivatives
    - Parameter Derivatives
    - Derivatives with Respect to Previous States
  - Gated Recurrent Unit (GRU) Networks: Forward Propagation
  - Gated Recurrent Unit (GRU) Networks: Backward Propagation
- Course 4-2: Word Embeddings
  - Traditional Word Representation Approaches
    - One-Hot Encoding
    - Bag-of-Words (BoW)
    - TF-IDF
    - Limitations of Traditional Word Representation Approaches
  - Learning Word Embeddings
    - Similarity Functions
    - Embedding Matrix
    - Learning Word Embeddings: An Intuitive Approach
  - Case Study: Word2vec
    - Continuous Bag-of-Words (CBOW) Model
    - Skip-Gram Model
    - Problem at the SoftMax Layer
    - Negative Sampling
    - Selection of Samples
  - Case Study: GloVe
  - Properties of Word Embeddings: Analogical Reasoning
- Course 4-3: Beam Search & BLEU Score
  - Encoder-Decoder Architecture
  - Beam Search
    - Numerical Stability & Length Normalisation
    - Error Analysis in Beam Search
  - BLEU Score
- Course 4-4: Attention Mechanism & Transformer
  - Attention Mechanism
  - The Transformer Architecture: A First Look
  - Self-Attention
    - Multi-Head Attention
    - Masked Self-Attention
  - Positional Encoding
  - Other Crucial Components in the Transformer Model
    - Feed Forward Network (FFN)
    - Residual Connections and Layer Normalization
  - Transformers: Wrapping Up

Course 1A: Binary Classifiers with Logistic Regression

The *Neural Networks and Deep Learning* course constitutes the first module of the *Deep Learning Specialization*.

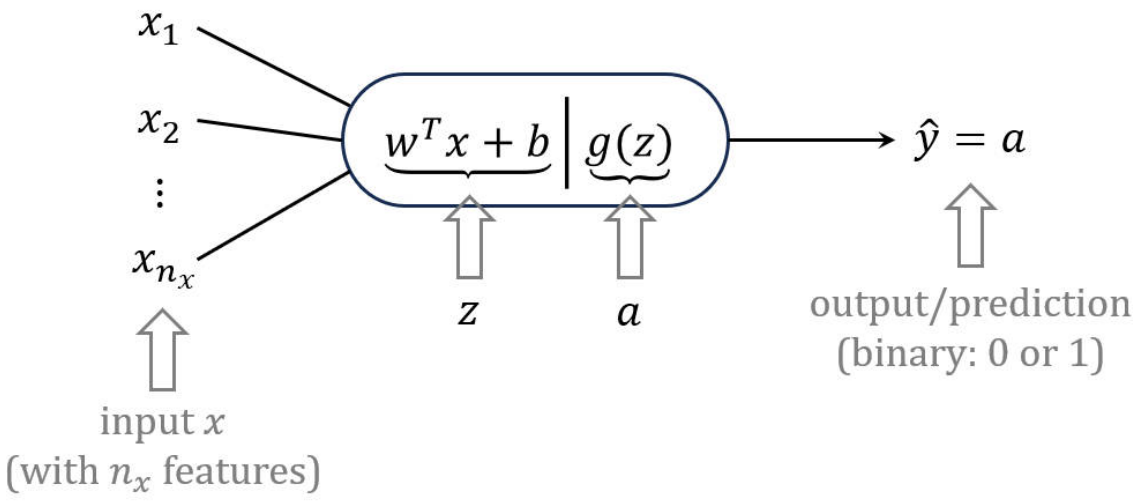
The entire course progresses from the simple to the deep - a progression that is both literal and metaphorical - teaching learners how to train and optimize a classification model based on the gradient descent algorithm. The video lectures devote the majority of the time to theoretical exposition, placing a particular emphasis on building intuitive understanding. Personally, I found it highly rewarding to thoroughly unravel the intricate details behind the forward and backward propagation formulas used in the simplified deep neural networks presented in the course. Consequently, the study notes I have compiled consist primarily of the theoretical sections from the course, supplemented by my own additional insights.

On a side note, I must truly express my admiration for the masterful proficiency with which practitioners in the field of machine learning utilize various parentheses and subscripts. Whether indexing layers, neurons, individual samples, or later on mini-batches within a neural network, the ability to devise such a wide variety of distinct notational schemes without repetition is truly mind-boggling. Furthermore, concepts that sound incredibly sophisticated - such as "backward propagation" and "vectorization" - have led me to marvel at the profound artistry involved in the nomenclature of the deep learning field. Yet, upon digging to the root of the matter, one discovers that these concepts are, in essence, merely the fundamental calculus principles of the chain rule for differentiation and matrix multiplication. However, when calculating the partial derivatives for that extensive set of loss functions, one can bypass the computationally expensive operation of matrix inversion. Instead, one simply needs to save the intermediate calculations from the forward propagation pass. Later, when working backward to compute the partial derivatives, these saved values can be retrieved. By performing nothing more than basic arithmetic - addition, subtraction, multiplication, and division - one can derive the necessary corrections for each step of gradient descent. When you really think about it, it's quite remarkable.

The practical component of the course is conducted within the Labs, where beginners are guided step-by-step in translating mathematical formulas into Python code. From implementing specific functions to encapsulating them into a complete training pipeline, the entire model-building process unfolds with remarkable smoothness—a testament to the immense thought and effort that Andrew Ng and the course production team undoubtedly poured into its design.

Forward Propagation

logistic regression is used to predict an output/prediction $\hat{y}$ from an input $\mathbf{x}$



- $z = \mathbf{w}^T\mathbf{x} + b$ (weighted sum of the input plus a bias term)
- $a = g(z)$ (activation function)
- $\hat{y} = a$ (predicted output)

where $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_x} \end{bmatrix}, \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n_x} \end{bmatrix}$ and $z, b \in \mathbb{R}$.

Estimation of Loss

we want to have $\hat{y} = y$ (i.e., prediction by model is the same as the desired output)

- **mean square error** (MSE): $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$.
- **cross-entropy**: $L(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$.
  - if $y = 1$, then $L(\hat{y}, y) = -\log \hat{y}$ minimised if $\hat{y} \to 1$
  - if $y = 0$, then $L(\hat{y}, y) = -\log(1 - \hat{y})$ minimised if $\hat{y} \to 0$

for classification tasks, cross-entropy loss is more preferable:

- strong penalty for highly confident wrong predictions
- measure of difference between probability distributions
- lead to convex loss function (with logistic activation)

  ensure gradient descent does not stuck in local minima

for all m samples, we can define **cost function** to be:

$$J(\mathbf{w}, b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)}) \right]$$

our objective is to find $\mathbf{w}^*$ and b^* in parameter space such that $J(\mathbf{w}, b)$ is minimised

Vectorisation

for a dataset with m samples, we introduce

$$\text{input: } X = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}_{n_x \times m}$$

$$\text{weights: } W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n_x} \end{bmatrix}_{n_x \times 1}$$

$$\text{bias: } b = \begin{bmatrix} b & b & \cdots & b \end{bmatrix}_{1 \times m}$$

then we can write:

$$Z = W^T X + b = \begin{bmatrix} z^{(1)} & z^{(2)} & \cdots & z^{(m)} \end{bmatrix}_{1 \times m}$$
$$A = \begin{bmatrix} g(z^{(1)}) & g(z^{(2)}) & \cdots & g(z^{(m)}) \end{bmatrix}_{1 \times m} = \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix}_{1 \times m}$$

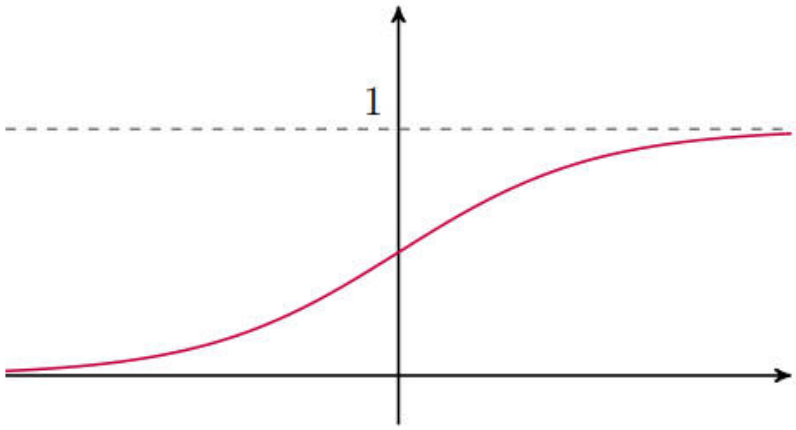
matrix multiplication $W^T X$ can be implemented with `np.dot`

to add bias b, Python can broadcast a scalar b to an array that matches the shape of $W^T X$

Activation Functions

Sigmoid

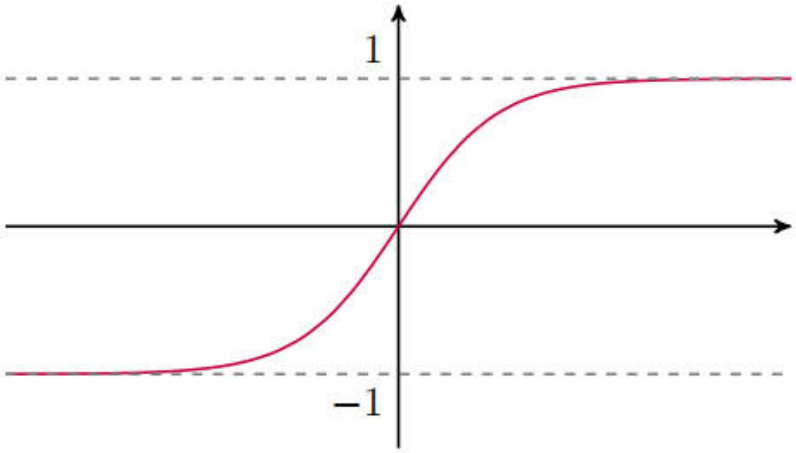
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



- limits: $\lim_{z \to +\infty} \sigma(z) = +1$ and $\lim_{z \to -\infty} \sigma(z) = 0$
- derivative: $\sigma'(z) = \dfrac{e^{-z}}{(1+e^{-z})^2} = \dfrac{1}{1+e^{-z}} \cdot \dfrac{e^{-z}}{1+e^{-z}} \Rightarrow \sigma'(z) = \sigma(z)(1-\sigma(z))$.
- limit of derivative: $\lim_{z \to \pm\infty} \sigma'(z) = 0$

Hyperbolic Tangent

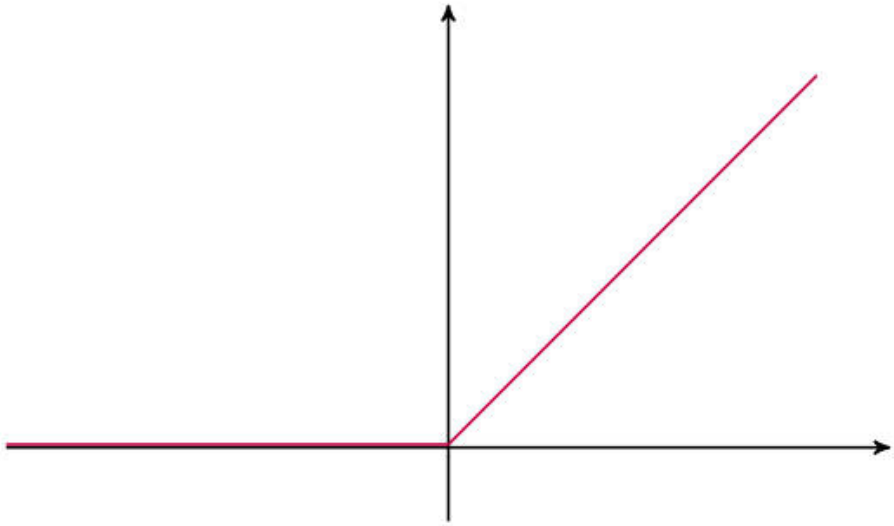
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



- limits: $\lim_{z \to \pm\infty} \tanh(z) = \pm 1$
- derivative: $\tanh'(z) - 1 - \tanh^2(z)$
- limit of derivative: $\lim_{z \to \pm\infty} \tanh'(z) = 0$

ReLU (Rectified Linear Unit)

$$\mathrm{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{for } z \geq 0 \\ 0 & \text{for } z < 0 \end{cases}$$

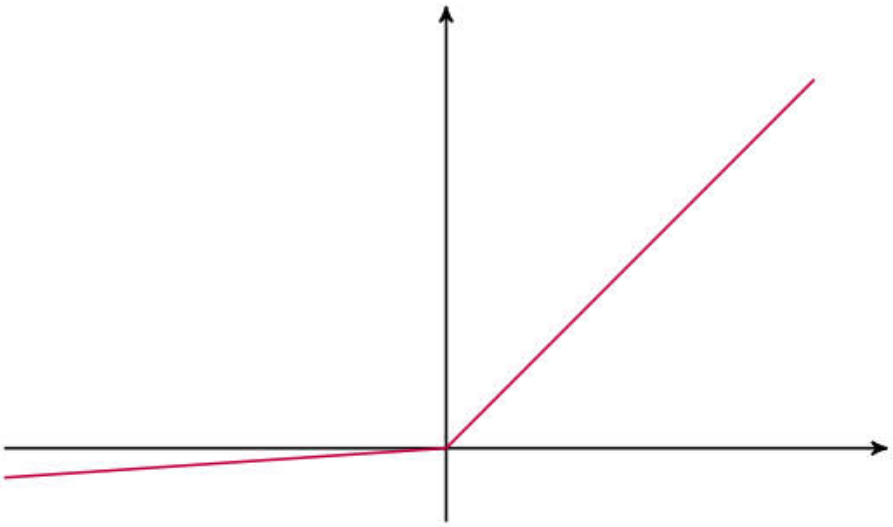


- derivative: $\mathrm{ReLU}(z) = \begin{cases} 1 & \text{for } z \geq 0 \\ 0 & \text{for } z < 0 \end{cases}$

  technically $\mathrm{ReLU}(z)$ is not defined at $z = 0$, but in practice $p(z = 0) \approx 0$

Leaky ReLU

$$\text{leaky ReLU} = \max(\beta, z) = \begin{cases} z & \text{for } z \geq 0 \\ \beta z & \text{for } z < 0 \end{cases} \quad \text{where } \beta \ll 1$$



How to Choose Activation Functions?

an empirical rule for choosing activation functions:

- output layer of binary classifier $\Rightarrow$ sigmoid function
- hidden layers in the network $\Rightarrow \mathrm{ReLU}, \tanh$

Gradient Descent and Backward Propagation

iteratively update parameters so gradient descent takes $J(\mathbf{w}, b)$ closer to its global minimum

$$\mathbf{w} := \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}} \qquad b := b - \alpha \frac{\partial J}{\partial b}$$

the hyperparameter α is the **learning rate**

Backward Propagation for Cross-entropy

the derivatives $\frac{\partial J}{\partial \mathbf{w}}$ and $\frac{\partial J}{\partial b}$ are computed using the chain rule

it turns out that these derivatives can be calculated using the cached values obtained during forward propagation

for cross-entropy loss:

$$\begin{aligned}\frac{\partial J}{\partial a^{(i)}} &= -\frac{1}{m}\frac{\partial}{\partial a^{(i)}}\left[y^{(i)}\log a^{(i)} + (1-y^{(i)})\log(1-a^{(i)})\right] \\ \Rightarrow \frac{\partial J}{\partial a^{(i)}} &= -\frac{1}{m}\left[\frac{y^{(i)}}{a^{(i)}} - \frac{1-y^{(i)}}{1-a^{(i)}}\right]\end{aligned}$$

suppose we use sigmoid as the activation function

$$\begin{aligned}\frac{\partial J}{\partial z^{(i)}} &= \frac{\partial J}{\partial a^{(i)}}\frac{\partial a^{(i)}}{\partial z^{(i)}} \\ &= -\frac{1}{m}\left[\frac{y^{(i)}}{a^{(i)}} - \frac{1-y^{(i)}}{1-a^{(i)}}\right] \times a^{(i)}(1-a^{(i)}) \\ &= -\frac{1}{m}\left[y^{(i)}(1-a^{(i)}) - (1-y^{(i)})a^{(i)}\right] \\ \Rightarrow \frac{\partial J}{\partial z^{(i)}} &= \frac{1}{m}\left(a^{(i)} - y^{(i)}\right)\end{aligned}$$

with $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)} + b$, we have

$$\begin{aligned}\frac{\partial J}{\partial b} &= \sum_{i=1}^{m}\frac{\partial J}{\partial z^{(i)}}\frac{\partial z^{(i)}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(a^{(i)} - y^{(i)}\right) \\ \frac{\partial J}{\partial w_k} &= \sum_{i=1}^{m}\frac{\partial J}{\partial z^{(i)}}\frac{\partial z^{(i)}}{\partial w_k} = \frac{1}{m}\sum_{i=1}^{m}\left(a^{(i)} - y^{(i)}\right)x_k^{(i)}\end{aligned}$$

by spirit of vectorisation, we can introduce

$$\begin{aligned}X &= \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}_{n_x \times m} \\ Y &= \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}_{1\times m} \\ A &= \begin{bmatrix} a^{(1)} & a^{(2)} & \cdots & a^{(m)} \end{bmatrix}_{1\times m}\end{aligned}$$

then we have:

$$\begin{aligned}db \equiv \frac{\partial J}{\partial b} &= \frac{1}{m} \times (\text{sum of elements of } A - Y) \\ dW \equiv \begin{bmatrix} \frac{\partial J}{\partial w_1} \\ \frac{\partial J}{\partial w_2} \\ \vdots \\ \frac{\partial J}{\partial w_{nx}} \end{bmatrix}_{n_x \times 1} &= \frac{1}{m}X(A-Y)^T\end{aligned}$$

these results can be implemented with the Python codes

$$\begin{aligned}db &= \frac{1}{m} * \text{np.sum}(A - Y) \\ dW &= \frac{1}{m} * \text{np.dot}(X, (A-Y).\text{T})).\end{aligned}$$

Implementation of Binary Classifier

initialisation of weights

```
w = np.random.randn(n_x, 1)
b = 0
```

- when initialising the weights, one can multiply W by a small number (e.g., multiply by 0.01) to keep away from the flat part of the sigmoid activation
- bias b is usually initialised to 0

optimisation (iteration through `for` loop):

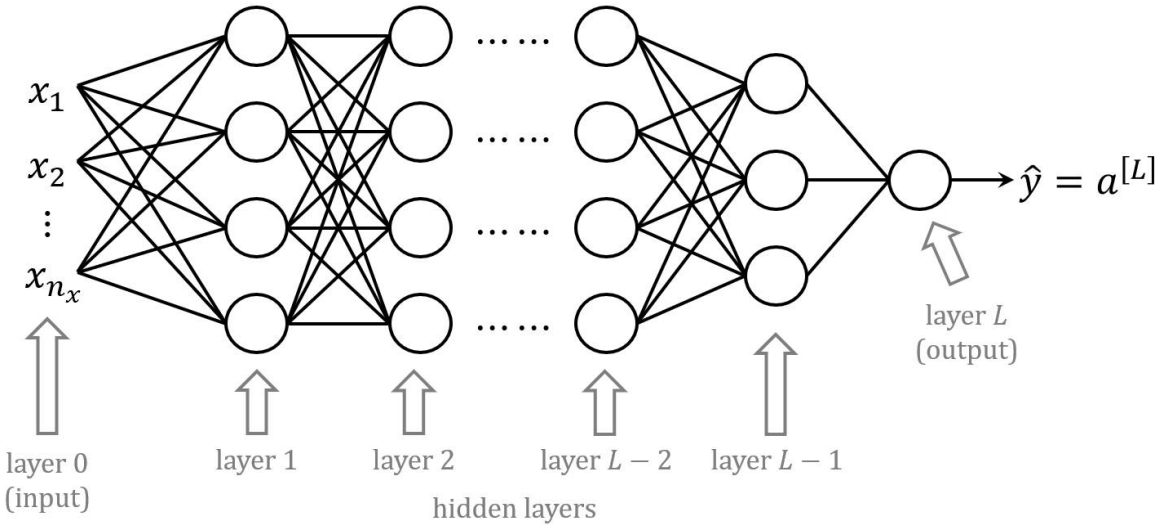
```
# forward propagation
Z = np.dot(W.T, X) + b
A = sigmoid(Z)

# backward propagation
dZ = 1/m * (A-Y)
dw = np.dot(X, dZ.T)
db = np.sum(dZ)

# gradient descent
w = w - learning_rate * dw
b = b - learning_rate * db
```

Course 1B: Multi-layer Neural Networks

Deep L-layer Neural Network



Notations

- $n^{[l]}$: number of neurons in layer l
- $a^{[l]} = g^{[l]}(z^{[l]})$: activations in layer l
- $W^{[l]}, b^{[l]}$: weights and biases for $z^{[l]}$
- $l = 0$: $a^{[0]} = X$ (input).
- $l = L$: $a^{[L]} = \hat{y}$ (predicted output).

Forward Propagation

at each layer l ($l = 1, 2, \cdots, L$), we want to compute an output $a^{[l]}$ from an input $a^{[l-1]}$:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

more explicitly, this is:

$$
\begin{bmatrix} z_1^{[l]} \\ z_2^{[l]} \\ \vdots \\ z_{n^{[l]}}^{[l]} \end{bmatrix} = \begin{bmatrix} \cdots & w_1^{[l]} & \cdots \\ \cdots & w_2^{[l]} & \cdots \\ & \vdots & \\ \cdots & w_{n^{[l]}}^{[l]} & \cdots \end{bmatrix} \begin{bmatrix} a_1^{[l-1]} \\ a_2^{[l-1]} \\ \vdots \\ a_{n^{[l-1]}}^{[l-1]} \end{bmatrix} + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix}
$$

$$
\begin{bmatrix} a_1^{[l]} \\ a_2^{[l]} \\ \vdots \\ a_{n^{[l]}}^{[l]} \end{bmatrix} = \begin{bmatrix} g^{[l]}(z_1^{[l]}) \\ g^{[l]}(z_2^{[l]}) \\ \vdots \\ g^{[l]}(z_{n^{[l]}}^{[l]}) \end{bmatrix}
$$

where the dimensions of the vectors and matrices are:

- $z^{[l]}$, $a^{[l]}$: $(n^{[l]} \times 1)$
- weight matrix $W^{[l]}$: $(n^{[l]} \times n^{[l-1]})$
- bias $b^{[l]}$: $(n^{[l]} \times 1)$
- $a^{[l-1]}$: $(n^{[l-1]} \times 1)$

Backward Propagation

at layer l ($l = L, L-1, \cdots, 2, 1$), we want to compute the derivatives $da^{[l-1]}, dW^{[l]}, db^{[l]}$ from the derivative values of $da^{[l]}$

Derivative for Neurons

for j^{th} neuron in layer l:

$$
\begin{aligned}
\frac{\partial J}{\partial z_j^{[l]}} &= \frac{\partial J}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = \frac{\partial J}{\partial a_j^{[l]}} \frac{\partial g^{[l]}(z_j^{[l]})}{\partial z_j^{[l]}} = \frac{\partial J}{\partial a_j^{[l]}} g^{[l]'}(z_j^{[l]}) \\
\Rightarrow \frac{\partial J}{\partial z^{[l]}} &= \frac{\partial J}{\partial a^{[l]}} g^{[l]'}(z^{[l]})
\end{aligned}
$$

this can be written as $dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$ where

- $dz^{[l]}, da^{[l]}, g^{[l]'}(z^{[l]})$ are all of the shape $(n^{[l]} \times 1)$
- "$*$" means element-wise multiplication

Derivative for Weights

for k^{th} element of the weights ($k = 1, 2, \cdots, n^{[l-1]}$) for j^{th} neuron ($j = 1, 2, \cdots, n^{[l]}$) in layer l:

$$
\begin{aligned}
\frac{\partial J}{\partial w_{jk}^{[l]}} &= \frac{\partial J}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial J}{\partial z_j^{[l]}} \frac{\partial}{\partial w_{jk}^{[l]}} \left(\sum_{k=1}^{n^{[l-1]}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]} \right) = \frac{\partial J}{\partial z_j^{[l]}} a_k^{[l-1]} \\
\Rightarrow \frac{\partial J}{\partial W^{[l]}} &= \frac{\partial J}{\partial z^{[l]}} {a^{[l-1]}}^T
\end{aligned}
$$

this can be written as $dW^{[l]} = dz^{[l]} {a^{[l-1]}}^T$ where

- $dW^{[l]}$ has dimensions $(n^{[l]} \times n^{[l-1]})$
- $dz^{[l]}$ has dimensions $(n^{[l]} \times 1)$
- ${a^{[l-1]}}^T$ has dimensions $(n^{[l-1]} \times 1)^T$

Derivative for Biases

for j^{th} neuron in layer l:

$$
\begin{aligned}
\frac{\partial J}{\partial b_j^{[l]}} &= \frac{\partial J}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial J}{\partial z_j^{[l]}} \frac{\partial}{\partial b_j^{[l]}} \left(\sum_{k=1}^{n^{[l-1]}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]} \right) = \frac{\partial J}{\partial z_j^{[l]}} \\
\Rightarrow \frac{\partial J}{\partial b^{[l]}} &= \frac{\partial J}{\partial z^{[l]}}
\end{aligned}
$$

this can be written as $db^{[l]} = dz^{[l]}$ where

- both $db^{[l]}$ and $dz^{[l]}$ are all of the shape $(n^{[l]} \times 1)$

Derivative for Activations in the Previous Layer

for k^{th} neuron in layer $l-1$:

$$\begin{aligned}
\frac{\partial J}{\partial a_k^{[l-1]}} &= \sum_{j=1}^{n^{[l]}} \frac{\partial J}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial a_k^{[l-1]}} \\
&= \sum_{j=1}^{n^{[l]}} \frac{\partial J}{\partial z_j^{[l]}} \frac{\partial}{\partial a_k^{[l-1]}} \left(\sum_{k'=1}^{n^{[l-1]}} w_{jk'}^{[l]} a_{k'}^{[l-1]} + b_j^{[l]} \right) \\
&= \frac{\partial J}{\partial z_j^{[l]}} w_{jk}^{[l]} \\
\Rightarrow \frac{\partial J}{\partial a^{[l-1]}} &= {W^{[l]}}^T \frac{\partial J}{\partial z^{[l]}}
\end{aligned}$$

this can be written as $da^{[l-1]} = {W^{[l]}}^T dz^{[l]}$

Backward Propagation Functions Wrap-up

for each layer $l = L, L-1, \cdots, 2, 1$:

- $dz^{[l]} = da^{[l]} * {g^{[l]}}'(z^{[l]})$
- $dW^{[l]} = dz^{[l]} {a^{[l-1]}}^T$
- $db^{[l]} = dz^{[l]}$
- $da^{[l-1]} = {W^{[l]}}^T dz^{[l]}$

Vectorisation with m Training Examples

with vectorisation, we can work with the entire dataset as a whole

we can introduce

$$A^{[l]} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ a^{[l](1)} & a^{[l](2)} & \cdots & a^{[l](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}_{n^{[l]} \times m}$$

$$W^{[l]} = \begin{bmatrix} \cdots & w_1^{[l]} & \cdots \\ \cdots & w_2^{[l]} & \cdots \\ & \vdots & \\ \cdots & w_{n^{[l]}}^{[l]} & \cdots \end{bmatrix}_{n^{[l]} \times n^{[l-1]}}$$

$$b^{[l]} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix}_{n^{[l]} \times 1}$$

where

- $A^{[0]} = X = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \end{bmatrix}$ is the input data
- $A^{[L]} = \hat{Y} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \cdots & \hat{y}^{(m)} \end{bmatrix}$ is the predicted output
- $Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$ is the desired output

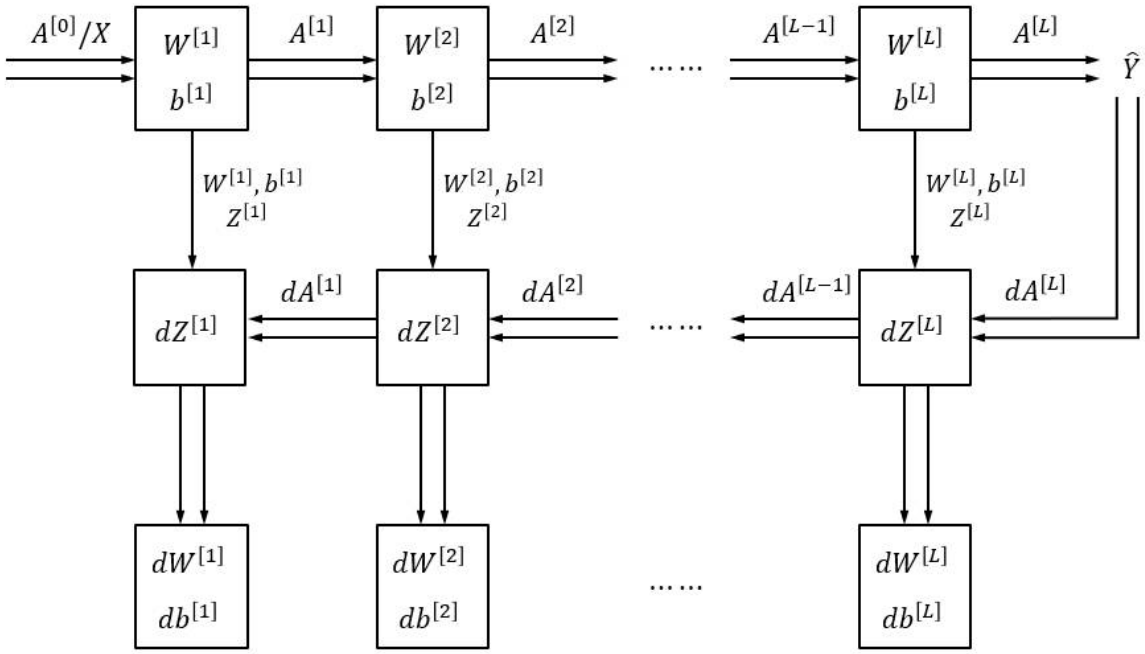
then for forward propagation at layer l:

$$\begin{aligned}
Z^{[l]} &= W^{[l]} A^{[l-1]} + b^{[l]} \\
A^{[l]} &= g^{[l]}(Z^{[l]})
\end{aligned}$$

for backward propagation at layer l:

$$
\begin{aligned}
dZ^{[l]} &= dA^{[l]} * g^{[l]'}(Z^{[l]}) \\
dW^{[l]} &= dZ^{[l]} A^{[l-1]T} \\
db^{[l]} &= \sum_{i=1}^{m} dZ^{[l](i)} \\
dA^{[l-1]} &= {W^{[l]}}^T dZ^{[l]}
\end{aligned}
$$

Typical Architecture of a Deep Neural Network



in this course, we use:

- ReLU/tanh as activation functions for hidden layers
- sigmoid as activation function for the output layer
- cross-entropy loss as the cost function

in this case, for the output layer, we have:

$$
\begin{aligned}
dZ^{[L]} &= dA^{[L]} * g^{[L]'}(Z^{[L]}) \\
&= -\frac{1}{m} \frac{\partial}{\partial A^{[L]}} \sum_{i=1}^{m} \left[Y^{(i)} \log A^{(i)} + (1 - Y^{(i)}) \log(1 - A^{(i)}) \right] * \sigma'(Z^{[l]}) \\
&= -\frac{1}{m} \left[\frac{Y}{A^{[L]}} - \frac{1 - Y}{1 - A^{[L]}} \right] * A^{[L]} * (1 - A^{[L]}) \\
\Rightarrow dZ^{[L]} &= \frac{1}{m} \left(A^{[L]} - Y \right)
\end{aligned}
$$

this gives the starting point for backward propagation for the entire network

Course 2: Hyper-parameter Tuning, Regularization and Optimization

Building upon the foundation of the first course, the second course covers a wide range of optimization techniques, including—but not limited to:

- **regularisation methods** for mitigating over-fitting issues
- optimisation algorithms like **momentum**, **RMSProp** and **Adam** that are designed to accelerate the convergence of gradient descent
- **normalisation schemes** that facilitate faster convergence

- weight **initialisation schemes** tailored to different activation functions
- generalisation from binary classifiers to **multi-class classifiers**

The study notes I have compiled continue to focus primarily on the theoretical component of the course, as well as various mathematical derivations I have added myself.

The second module of the course once again reinforced my impression that the naming of concepts in the field of deep learning is truly an art form: the Momentum optimization algorithm, designed to mitigate the impact of noise, is fundamentally indistinguishable from the Moving Average Filtering technique — a common technique in digital signal processing. Similarly, the structure of the Softmax activation function bears a striking resemblance to the Boltzmann distribution in statistical mechanics. Indeed, once upgraded with a temperature coefficient, it becomes absolutely identical. As for the operations introduced in this module — specifically those found in Normalization, Initialization, and RMSProp — they amount to nothing more than standard mean-variance normalization procedures borrowed from statistics. That being said, the sheer ingenuity involved in conceiving the application of these techniques to resolve the difficulties inherent in the gradient descent optimization process is, without a doubt, truly brilliant and worthy of the highest acclaim.

Bias-Variance Trade-off

intuition of the notion of bias and variance:

- low training error / low test error $\Rightarrow$ low bias / low variance

  the model gives a good representation of distribution for training data

  it also generalises well (not sensitive to fluctuations in training data)
- low training error / high test error $\Rightarrow$ low bias / high variance

  the training data might be overfit, the model does not generalise well
- high training error / similar test error $\Rightarrow$ high bias / low variance

  the model might not have sufficient complexity to capture the data distribution

Derivation of the Bias–Variance Decomposition

suppose each input x is mapped into an output y as

$$y = f(x) + \epsilon$$

where

- f is a deterministic function that we want to model
- ϵ epsilon is error in data with $E(\epsilon) = 0$ and $\mathrm{Var}(\epsilon) = \sigma^2$

the model makes a prediction

$$\hat{y} = \hat{f}(x)$$

for output y, we have:

$$\begin{aligned} E(y) &= E(f + \epsilon) = E(f) + \cancelto{0}{E(\epsilon)} = f \\ \mathrm{Var}(y) &= E\left([y - E(y)]^2\right) = E\left([(f + \epsilon) - f]^2\right) = E(\epsilon^2) = \sigma^2 \end{aligned}$$

for model error, we have

$$\begin{aligned} E\left((y - \hat{y})^2\right) &= E\left(y^2 + \hat{f}^2 - 2y\hat{f}\right) \\ &= E(y^2) + E(\hat{f}^2) - 2E(y\hat{f}) \\ &= \mathrm{Var}(y) + E(y)^2 + \mathrm{Var}(\hat{f}) + E(\hat{f})^2 - 2f \cdot E(\hat{f}) \\ &= \mathrm{Var}(y) + \mathrm{Var}(\hat{f}) + \left[f^2 - 2f \cdot E(\hat{f}) + E(\hat{f})^2\right] \\ &= \mathrm{Var}(y) + \mathrm{Var}(\hat{f}) + \left(f - E(\hat{f})\right)^2 \\ \Rightarrow E\left((y - \hat{y})^2\right) &= \sigma^2 + \mathrm{Var}(\hat{f}) + \mathrm{Bias}(\hat{f})^2 \end{aligned}$$

so we can see that the expected squared difference between the output and the prediction $\hat{f}$ consists of three parts:

- σ^2 being a measure of the uncertainty in the outputs, this is the inherent noise in the data which cannot be eliminated by any modelling technique regardless of how well the model is trained
- $\mathrm{Var}(\hat{f})$ represents the uncertainty in the mapping function, this measures how much the prediction $\hat{f}(x)$ varies across different training datasets
- $\mathrm{Bias}(\hat{f})^2$ being the error arising from erroneous assumptions in the learning algorithm, which may cause the average prediction over different training sets to deviated from the true value

Recipe for Machine Learning

- if there is high bias
  - try larger networks with more layers and more neurons
  - try a different model that is more suitable for the data
  - try different or advanced optimisation algorithms
- if there is high variance
  - try with more training data
  - try regularisation methods
  - try a different model that is more suitable for the data

Regularisation

> idea: penalise large weights to make the neural network less likely to overfit

L_2-regularisation

cost function with L_2-regularisation becomes:

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|W^{[l]}\|_2^2$$

where

- λ is the regularisation hyperparameter
- $\|W^{[l]}\|_2^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}\left(w_{ij}^{[l]}\right)^2$ is the sum of squares for all weights

this introduces an extra term for $\dfrac{\partial J}{\partial W^{[l]}}$:

$$\begin{aligned}\frac{\partial J}{\partial w_{ij}^{[l]}} &= (\cdots) + \frac{\lambda}{2m}\frac{\partial}{\partial w_{ij}^{[l]}}\left(w_{ij}^{[l]}\right)^2 = (\cdots) + \frac{\lambda}{2m}w_{ij}^{[l]} \\ \Rightarrow \frac{\partial J}{\partial W^{[l]}} &= (\cdots) + \frac{\lambda}{m}W^{[l]} \quad \text{or} \quad dW^{[l]} = (\cdots) + \frac{\lambda}{m}W^{[l]}\end{aligned}$$

note that during gradient descent, the update rule for the weights becomes

$$W^{[l]} := W^{[l]} - \alpha \cdot dW^{[l]} = \left(1 - \frac{\alpha\lambda}{m}\right)W^{[l]} - \alpha(\cdots)$$

the first term shows a weight decay for each iteration step, forcing the model to keep the weights small

L_1-regularisation

cost function with L_1-regularisation becomes:

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|W^{[l]}\|_1$$

where

- λ is the regularisation hyperparameter
- $\|W^{[l]}\|_1 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} |w_{ij}^{[l]}|$ is the sum of absolute values for all weights

in practice, L_1-regularisation could set many weights to zero, which makes the neural network simpler so less prone to over-fitting problems

Other Regularisation Methods

- **dropout regularisation**

  for each training example, some neurons or weights are randomly eliminated based on a probability, leading to a smaller network

  the intuition is that the network should not rely on any one feature as it may go away at random, so the network should spread out weights

  this tends to produce a similar effect to shrinking $\|W\|_2^2$
- **data augmentation**

  this is a widely used method in computer vision where input size is big but there are almost never enough training samples

  the idea is to augment the training set by flipping / shifting / cropping / rotating images
- **early stopping**

  at some iteration step, development set error would stop decreasing and but start to increase, signalling over-fitting

  so we can stop training and retrieve weights from the epoch when the performance on the development set was the best

Initialisation

Normalising Inputs

two-step procedure for input normalisation:

- subtract the mean: $X := X - \mu = X - \frac{1}{m} \sum_{i=1}^{m} X^{(i)}$
- normalise the variance: $X := \frac{X}{\sigma} = \frac{X}{\frac{1}{m} \left(\sum_{i=1}^{m} {X^{(i)}}^2 \right)^{\frac{1}{2}}}$

raw input data is converted into more consistent scales

shape of the cost function in parameter space becomes more symmetrical

this often allows the model to converge to an optimal solution faster

Weight Initialisation

> idea: poor initialisation can lead to vanishing or exploding gradients, which slows down the optimisation process

better initialisations that are often used include

- for $\tanh$
  - **Xavier initialisation**: $W^{[l]} \sim N(0, 1) \times \sqrt{\frac{1}{n^{[l-1]}}}$
  - Bengio (et al.) initialisation: $W^{[l]} \sim N(0, 1) \times \sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
- for ReLU
  - **He initialisation**: $W \sim N(0, 1) \times \sqrt{\frac{2}{n^{[l-1]}}}$

where $N(0, 1)$ is the Gaussian distribution with zero mean and unit variance

in Python, this can be implemented by

$$W^{[l]} = \text{np.random.randn}(n^{[l]}, n^{[l-1]}) * \text{np.sqrt}(\cdots)$$

Optimisation Algorithms

Mini-batch Gradient Descent

> idea: update parameters with mini batches (portions of training set) for faster convergence

mini-batch sizes

- if size of mini-batch $= m \Rightarrow$ regular gradient descent
- if size of mini-batch $= 1 \Rightarrow$ stochastic gradient descent (SGD)
- if size of mini-batch $= m_b$ where $1 < m_b < m$

  updates would be less noisy than SGD but can also take advantage of vectorisation
- typical mini-batch sizes include 32, 64, 128, or 256 to better fit in CPU or GPU memory

applying mini-batch gradient descent

implementation involves

- create a randomly shuffled version of the training set (X, Y)
- partition the shuffled (X, Y) it into mini-batches $(X^{\{1\}}, Y^{\{1\}}), (X^{\{2\}}, Y^{\{2\}}), \cdots, (X^{\{t\}}, Y^{\{t\}})$

  (note that size of the last mini-batch can be smaller than m_b)

Gradient Descent with Momentum

> idea: path taken by mini-batch gradient descent might oscillate towards convergence, using the moving averages of the past gradients could smooth out such oscillations

gradient descent with momentum algorithm is analogous to moving average filters that reduce random noise in digital signal processing

on iteration t:

- compute dW, db on current mini-batch
- take move averages of past gradients

$$\begin{aligned} V_{dw} &:= \beta V_{dw} + (1-\beta) dW \\ V_{db} &:= \beta V_{db} + (1-\beta) db \end{aligned}$$

- update parameters with dW, db replaced by weighted averages:

$$\begin{aligned} W &:= W - \alpha V_{dw} \\ b &:= b - \alpha V_{db} \end{aligned}$$

bias correction for low initial values

if we initialise with $v_0 = 0$, then first few averages $v_1, v_2, \cdots$ would be underestimated due to heavy influence of the starting point

let v_t be the moving average of variable $\theta_t (t = 1, 2, \cdots)$, then

$$\begin{aligned} v_t &= \beta v_{t-1} + (1-\beta)\theta_t \\ v_t &= \beta^2 v_{t-2} + \beta(1-\beta)\theta_{t-1} + (1-\beta)\theta_t \\ &\vdots \\ v_t &= \beta^t \cancel{v_0}^{0} + \beta^{t-1}(1-\beta)\theta_1 + \beta^{t-2}(1-\beta)\theta_2 + \cdots + (1-\beta)\theta_t \\ v_t &= (1-\beta)\left[\beta^{t-1}\theta_1 + \beta^{t-2}\theta_2 + \cdots + \beta\theta_{t-1} + \theta_t\right] \end{aligned}$$

this shows an exponetially weighted average with a sum of weights being

$$(1-\beta)\left[\beta^{t-1} + \beta^{t-2} + \cdots + \beta + 1\right] = 1 - \beta^t$$

but a weighted average should have normalised weights, so we should correct the moving average for v_t by:

$$V_t^{\text{corrected}} = \frac{V_t}{1-\beta^t}$$

RMSProp (Root Mean Square Propagation)

> idea: hope to accelerate steps in directions with small oscillations and slow down steps in directions with large oscillations

on iteration t:

- compute dW, db on current mini-batch
- take move averages of variances of dW and db

$$S_{dw} := \beta S_{dw} + (1-\beta) dW^2$$
$$S_{db} := \beta S_{db} + (1-\beta) db^2$$

- update parameters with RMSProp for gradient descent

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$$
$$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

Adam Optimisation Algorithm

> idea: combine gradient descent with momentum and RMSProp

on iteration t:

- compute dW, db on current mini-batch
- calculates moving averages for both momentum (V) and RMSProp (S)

$$V_{dw} := \beta_1 V_{dw} + (1-\beta_1) dW$$
$$V_{db} := \beta_1 V_{db} + (1-\beta_1) db$$
$$S_{dw} := \beta_2 S_{dw} + (1-\beta_2) dW^2$$
$$S_{db} := \beta_2 S_{db} + (1-\beta_2) db^2$$

- apply bias correction to the moving averages

$$V_{dW}^{\text{corrected}} = \frac{V_{dW}}{1-\beta_1^t} \qquad V_{db}^{\text{corrected}} = \frac{V_{db}}{1-\beta_1^t}$$
$$S_{dW}^{\text{corrected}} = \frac{S_{dW}}{1-\beta_2^t} \qquad S_{db}^{\text{corrected}} = \frac{S_{db}}{1-\beta_2^t}$$

- update parameters using gradient descent

$$W := W - \alpha \frac{V_{dW}^{\text{corrected}}}{\sqrt{S_{dW}^{\text{corrected}}}}$$
$$b := W - \alpha \frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}}}}$$

there are three hyperparameters for Adam

- α: learning rate
- β_1 for momentum
- β_2 for RMSProp

in practice, we can include a small number $\epsilon \approx 0$ (e.g., $\epsilon = 10^{-8}$) to avoid division-by-zero error

$$W := W - \alpha \frac{V_{dW}^{\text{corrected}}}{\sqrt{S_{dW}^{\text{corrected}} + \epsilon}}$$
$$b := W - \alpha \frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}} + \epsilon}}$$

Hyperparameter Tuning

Examples of Hyperparameters

in order of importance, examples of hyperparameters include

- learning rate α
- β-parameter for momentum
- mini-batch size
- number of hidden units and number of layers
- regularisation parameter λ
- β-parameter for RMSProp
- choice of activation functions
- $\cdots$

rule of thumb for picking hyperparameters

use a logarithmic scale to search for hyperparameters

suppose we want to search optimal α^* within the range $m \leq \alpha \leq M$

$$\begin{aligned} \text{low} &= \log_{10} m \\ \text{high} &= \log_{10} M \\ r &= \text{low} + \text{np.random.rand}() * (\text{high} - \text{low}) \\ \alpha &= 10^r \end{aligned}$$

Batch Normalisation

batch normalisation means normalising not only the inputs but also intermediate activations to have zero mean and unit variance

batch normalisation can make hyperparameter search easier and the neural network more robust

algorithm:

- for activations: $Z^{[l]} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ Z^{[l](1)} & Z^{[l](2)} & \cdots & Z^{[l](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}$
- compute mean: $\mu^{[l]} = \dfrac{1}{m} \displaystyle\sum_{i=1}^{m} Z^{[l](i)}$
- compute variance: ${\sigma^{[l]}}^2 = \dfrac{1}{m} \displaystyle\sum_{i=1}^{m} \left(Z^{[l](i)} - \mu^{[l]} \right)^2$
- normalisation: $Z_{\text{norm}}^{[l](i)} = \dfrac{Z^{[l](i)} - \mu^{[l]}}{\sigma^{[l]}}$

we can further set

$$\tilde{Z}^{[l](i)} = \gamma^{[l]} Z_{\text{norm}}^{[l](i)} + \beta^{[l]}$$

where γ and β are learnable parameters of the model, this allows $\tilde{Z}^{[l]}$ to fit other mean or variance

at test time, we can use weighted averages of $\mu^{[l]}$ and $\sigma^{[l]}$ across mini-batches as an estimation

Multi-class Classification

for C classes, the output layer needs $n^{[L]} = C$ neurons, and each neuron $a_j^{[L]} = \hat{y}_j$ predicts the probability of an example belonging to the j^{th} class

SoftMax Activation

softmax activation for output layer:

$$a_j^{[L]} = \frac{e^{Z_j^{[L]}}}{\sum_{i=1}^{C} e^{Z_i^{[L]}}}$$

this can be implemented as

$$t = \text{np.exp}(Z^{[L]})$$
$$A^{[L]} = \frac{t}{\text{np.sum(t)}}$$

cost function used with softmax is

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{C} y_j^{(i)} \log \hat{y}_j^{(i)}$$
$$J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{C} y_j^{(i)} \log a_j^{[L](i)}$$

Backward Propagation for $dZ^{[l]}$

for simplicity, let's drop the $[L]$ / (i) indices for now

$$\begin{aligned}
\frac{\partial J}{\partial Z_j} &= \sum_{k=1}^{C} \frac{\partial J}{\partial a_k} \frac{\partial a_k}{\partial Z_j} \\
&= -\frac{1}{m} \sum_{k=1}^{C} \frac{\partial}{\partial a_k} (y_k \log a_k) \frac{\partial}{\partial Z_j} \left(\frac{e^{Z_k}}{\sum_{i=1}^{m} e^{Z_i}} \right) \\
&= -\frac{1}{m} \sum_{k=1}^{C} \frac{y_k}{a_k} \left[\frac{-e^{Z_k} \cdot e^{Z_j}}{\left(\sum_i e^{Z_i}\right)^2} + \delta_{jk} \frac{e^{Z_j}}{\sum_i e^{Z_i}} \right] \\
&= -\frac{1}{m} \sum_{k=1}^{C} \frac{y_k}{a_k} [-a_k \cdot a_j + \delta_{jk} \cdot a_j] \\
&= -\frac{1}{m} \sum_{k=1}^{C} (-y_k a_j + \delta_{jk} y_k) \\
&= \frac{1}{m} \underbrace{\left(\sum_{k=1}^{C} y_k \right)}_{1} a_j - \frac{1}{m} y_j \\
\Rightarrow \frac{\partial J}{\partial Z_j} &= \frac{1}{m} (a_j - y_j)
\end{aligned}$$

recovering indices, this is:

$$\frac{\partial J}{\partial Z_j^{[L](i)}} = \frac{1}{m} (a_j^{[L](i)} - y_j^{(i)})$$

in vectorised form, we have:

$$dZ^{[L]} = \frac{1}{m} \left(A^{[L]} - Y \right)$$

which takes the same form as the binary classifier

with $dZ^{[L]}$ as the starting point, the rest of the backward propagation steps for $dW^{[L]}$, $db^{[L]}$ and $dA^{[L]}$ could proceed in literally the same way as before (see note C1-B)

SoftMax with Temperature

it is possible to control the randomness and flatness of the output probability distribution by a temperature parameter T

$$a_j^{[L]} = \frac{e^{Z_j^{[L]}/T}}{\sum_{i=1}^{C} e^{Z_i^{[L]}/T}}$$

larger values of T produce a softer probability distribution over classes, making less probable options more likely

this results in less confidence and more diversity in predictions (also mistakes)

Course 3-1: Convolutional Neural Network Basics

I felt that this is the most challenging week in the Deep Learning Specialization so far: 2 hours of video lectures, plus 2 programming assignments, plus the time I spent on writing up the notes, it took me at least 10 hours on the basics of CNN.

In particular, we need to implement the **convolutional filters** and **pooling layers** using `numpy` only in the first programming assignment. Despite the guidance provided by the instructors, it was still very challenging to figure out every single detail of the computations and rewrite them into Python codes.

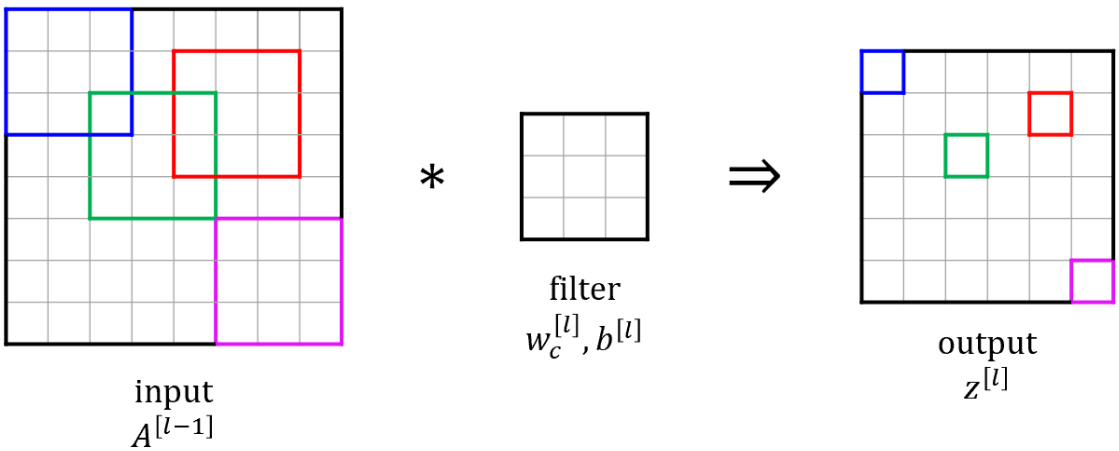
In the optional part of the programming assignment, the instructors encouraged the learners to try implementing the **backward propagation** computations for the convolutional layer and the pooling layers. A bunch of equations were given as a reference without proof. As a former theoretical physics student, my DNA would not allow me to take these unjustified equations for granted. I believe I had figured out the mathematics of nasty partial derivatives after some work, and I was satisfied to organize my derivation drafts into these notes while it was still fresh in my mind. The notations might be a bit messy, but I guess that is the way it is.

The good news is that we do not need to reinvent the wheel in the later part of the course. From Week 2, we will be using Tensorflow to build our own deep learning neural networks. Using the Keras Sequential API to build things up, sounds really awesome, isn't it?!

Convolutional Filtering

Convolutional Filters

> Idea: use a **sliding window** to detect specific features in a small portion of data such as images



For each slice:

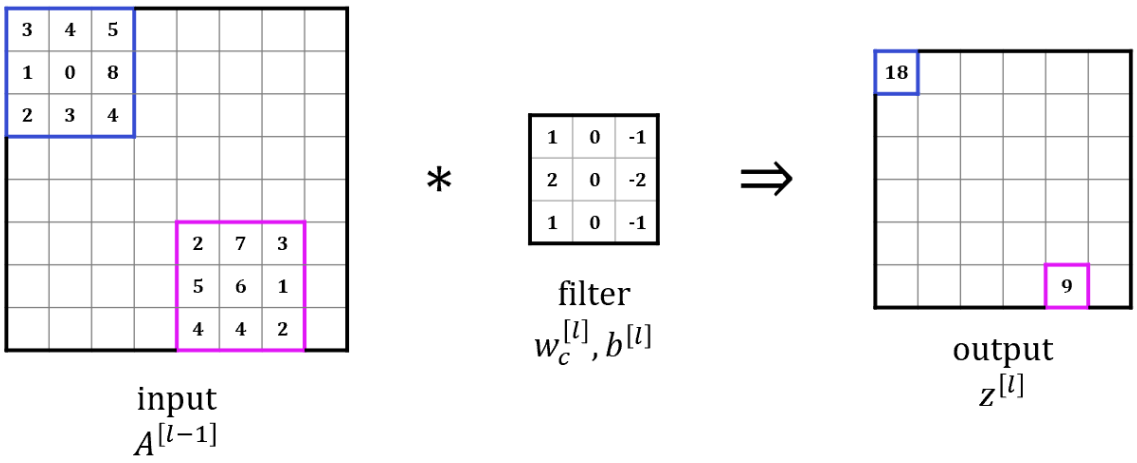
$$z_{hw}^{[l]} = \sum_{m=1}^{f}\sum_{n=1}^{f} w_c^{[l]}(m,n) \cdot A_{\text{slice},hw}^{[l-1]}(m,n) + b^{[l]}$$

where

- $z_{hw}^{[l]}$ is the output value of $z^{[l]}$ in h^{th} row and w^{th} column

- summation is performed over all elements of the filter of size $f \times f$
- $w_c^{[l]}(m,n) \cdot A_{\text{slice},hw}^{[l-1]}(m,n)$ is an element-wise multiplication between the elements
- $b^{[l]}$ is a bias term added after the summation

Let's illustrate how a convolutional filter works with an example.



$$\square = 1 \times 3 + 0 \times 4 + (-1) \times 5 + 2 \times 1 + 0 \times 0 + (-2) \times 8 + 1 \times 2 + 0 \times 3 + (-1) \times 4 = -18$$
$$\square = 1 \times 2 + 0 \times 7 + (-1) \times 3 + 2 \times 5 + 0 \times 6 + (-2) \times 1 + 1 \times 4 + 0 \times 4 + (-1) \times 2 = 9$$

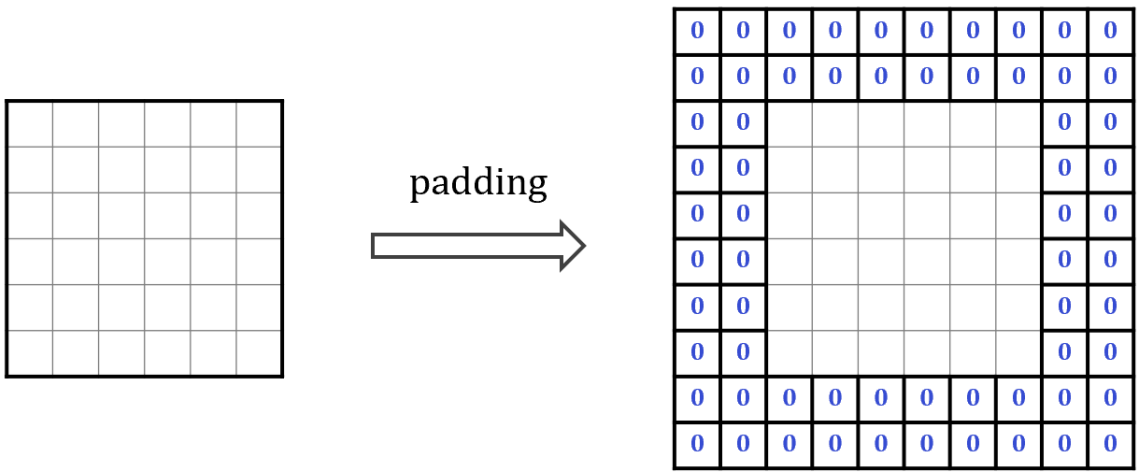
Why Convolutional Operation?

Convolutional operation is chosen as the foundational building block of CNN for many reasons, including

- **Edge detection**: useful for identifying spatial patterns (edges, shapes, textures, etc.) in computer vision
  For example, $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$ detects vertical edges, $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$ detects horizontal edges.
- **Parameter sharing**: feature detector in one part of the image may be useful in detecting the same feature in other parts as well
- Sparsity of connection: in each layer, each neuron only looks at a local patch of the previous layer, so each output value depends only on a small number of inputs

Padding

> Idea: adding extra pixels (usually zeros) to the borders of the input image before convolution



If padding amount is p, then the dimensions of data would be increased from the original $n \times n$ to $(n+2p) \times (n+2p)$ after padding.

The reasons for padding include:

- preserve spatial dimensions, i.e., avoid shrinking the output
- allow pixels near edges and corners to be processed equally

Common types of padding include:

- **valid padding** (no padding)

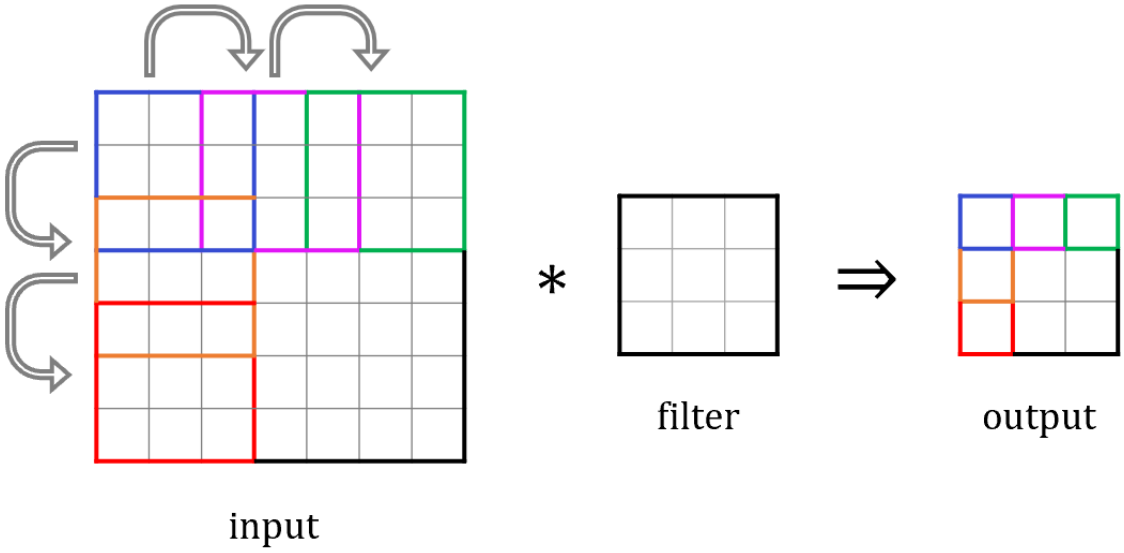
  dimensions of the output would be smaller after convolution
- **same padding**

  output image has the same dimensions as the input image

  we need $n + 2p - f + 1 = n$, so $p_{\text{same}} = \frac{f-1}{2}$ for odd f

Strided Convolutions

> Idea: allow the filter to move by more than one pixel at a time, this helps to reduce the output size, hence reduces computational cost



Convolution over Volumes

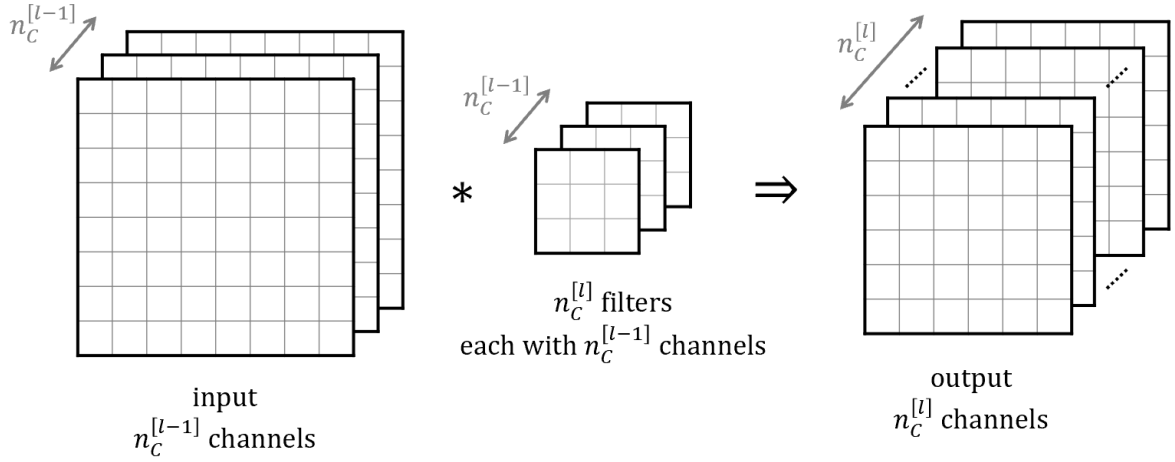
Input data may have more than one channel like three-channel RGB images. To allow the filter to extract spatial features across all channels, we need 3D convolutional filters to span the full depth of the input.

We can further use multiple filters of convolutional operation to extract multiple features at the same time. By stacking the results together, we produce an output with multiple channels.

Summary of Convolutional Filtering (One Layer)

Suppose at the l^{th} layer, we have

- input data with the dimensions $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$
- $n_C^{[l]}$ of filters each of size $f^{[l]} \times f^{[l]}$, padding of $p^{[l]}$, and stride of $s^{[l]}$



Then the dimensions of the output would be $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$, where

$$n_H^{[l]} = \left\lfloor \frac{n_H^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$
$$n_W^{[l]} = \left\lfloor \frac{n_W^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

Pooling

Pooling layers are used to reduce the spatial dimensions of input data, which decreases the number of parameters and computation required, speeding up training.

Typically pooling layers are calculating either the maximum (**Max Pooling**) or the average (**Average Pooling**) value of the elements within the sliding window.

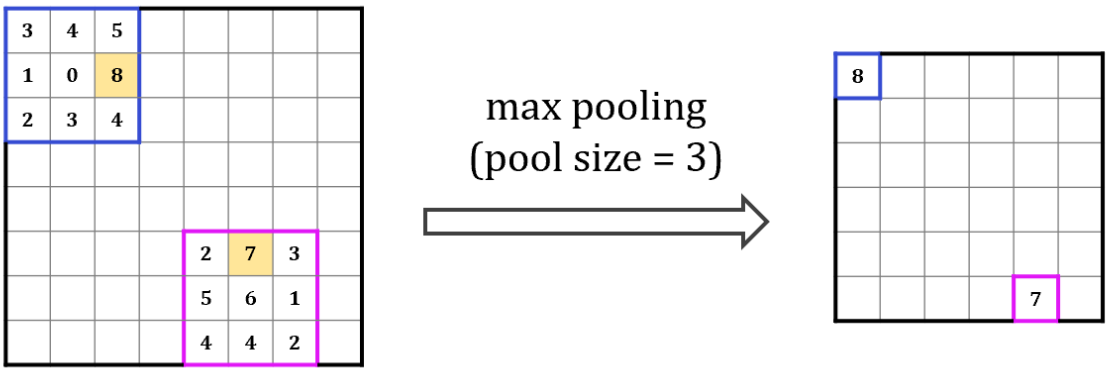
Max Pooling

> Idea: pick out the maximum value within the sliding window

For each slice:

$$P_{hw}^{[l]} = \max\left(A_{\text{slice},hw}^{[l-1]}(m,n)\right) \quad \text{for } m,n = 1,2,\cdots,f$$

An self-explanatory example is shown below:



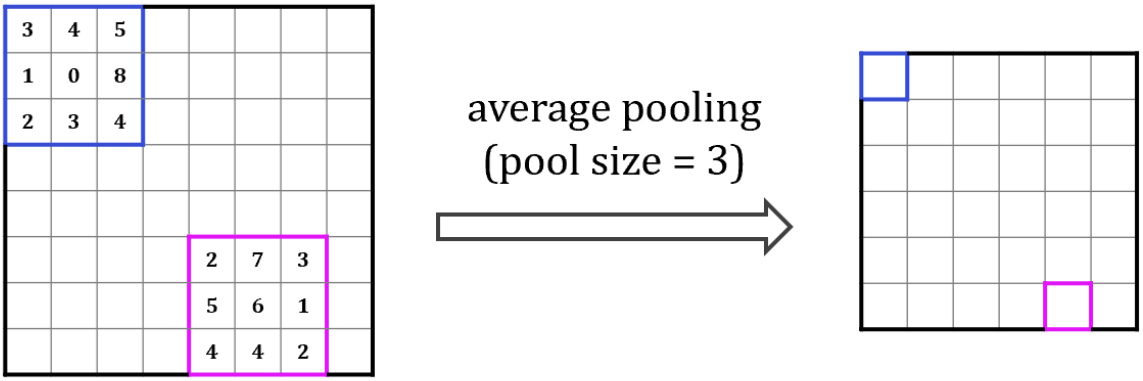
Average Pooling

> Idea: store the average value of the elements within the sliding window

For each slice:

$$P_{hw}^{[l]} = \frac{1}{f \times f} \sum_{m=1}^{f} \sum_{n=1}^{f} A_{\text{slice},hw}^{[l-1]}(m,n)$$

An example is shown below:

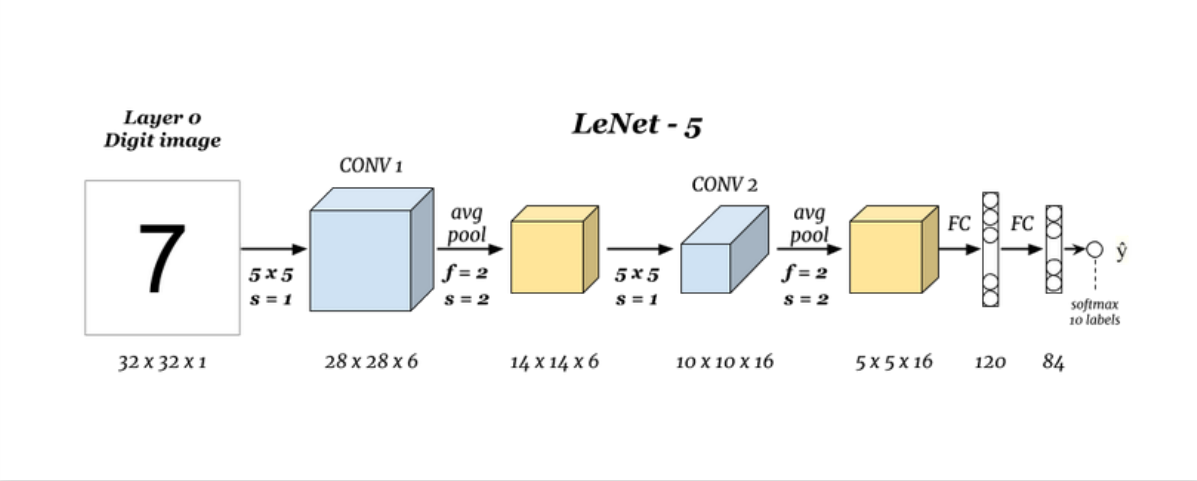


$$\square = \frac{1}{9}(3+4+5+1+0+8+2+3+4) = \frac{10}{3} \approx 3.33$$
$$\square = \frac{1}{9}(2+7+3+5+6+1+4+4+2) = \frac{34}{9} \approx 3.78$$

Convolutional Neural Networks

Typical CNN Architectures

The following figure shows the architecture of **LeNet** (LeNet-5), a small network introduced by and named for *Yann LeCun* initially for the purpose of recognizing handwritten digits in images. This model contains the basic modules of deep learning (convolutional layers, pooling layers, and full connected layers), and was among the first published CNNs to capture wide attention for its performance on computer vision tasks.



The first part of LeNet is a convolutional encoder consisting of two layers. Note that each convolutional layer and its subsequent pooling layer are usually considered as one layer in CNN community.

The second part of LeNet is a dense block consisting of three fully-connected (FC) layers. The activations are $A^{[l]} = g^{[l]}\left(W^{[l]}A^{[l-1]} + b^{[l]}\right)$ as before.

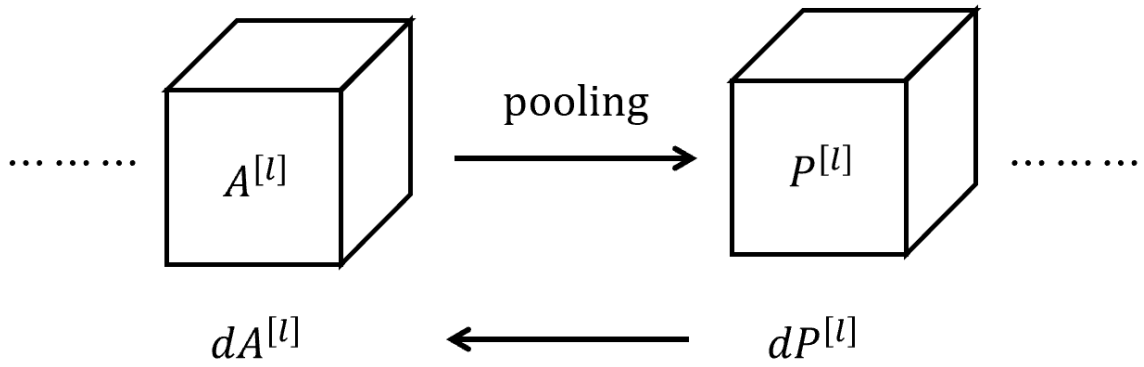
For a CNN architecture in general, it is worth pointing out that

- Successful CNN's usually consists of multiple layers of convolution and pooling.
- As one goes through the CNN, the size of the feature maps usually decreases ($n_H \downarrow, n_W \downarrow$) while the number of channels usually increases ($n_C \uparrow$)
- The hyperparameters of a CNN include the filter sizes $f^{[l]}$, strides $s^{[l]}$, the choice of type of pooling (max or average), etc.
- Weights of convolutional layers and weights of fully-connected layers are learnable parameters. They can be optimised through gradient descent methods.
- Fully-connected layers usually have the most parameters in the CNN.

Backward Propagation for CNN

Backward Pass for Pooling Layers

> Objective: compute $dA^{[l]}$ given $dP^{[l]}$.



Each element of $dA_{ij}^{[l]}$ receives a contribution from the $dP_{hw}^{[l]}$'s such that the corresponding value of $P_{hw}^{[l]}$ was calculated using a sliding window containing $A_{ij}^{[l]}$. So we can iterate over all $P_{hw}^{[l]}$'s and add up the associated contributions to each of the $dA_{ij}^{[l]}$'s to obtain $dA^{[l]}$.

Backward Pass for Max Pooling

For each $P_{hw}^{[l]} = \max\left(A_{\text{slice},hw}^{[l-1]}(m,n)\right)$, $dP_{hw}^{[l]}$ only contributes to one value of $A_{\text{slice},hw}^{[l]}$ in this window. We can introduce a mask matrix $M_{hw}^{[l]}$ that keeps track of where the maximum of $A_{\text{slice},hw}^{[l]}(m,n)$ occurs (i.e., $M_{hw}^{[l]}$ is a matrix with all zeros except one "1" at the position of the maximum entry), then we have:

$$dA_{\text{slice},hw}^{[l]} = M_{hw}^{[l]} * dP_{hw}^{[l]}$$

where both $dA_{\text{slice},hw}^{[l]}$ and $M_{hw}^{[l]}$ have the same shape as the sliding window.

Backward Pass for Average Pooling

For each $P_{hw}^{[l]} = \dfrac{1}{f \times f}\displaystyle\sum_{m=1}^{f}\sum_{n=1}^{f} A_{\text{slice},hw}^{[l-1]}(m,n)$, we have:

$$dA_{\text{slice},hw}^{[l]}(m,n) = \frac{1}{f \times f}\sum_{m=1}^{f}\sum_{n=1}^{f} dP_{hw}^{[l]} \quad \text{for all } m,n = 1,2,\cdots,f$$

where each of $dA_{\text{slice},hw}^{[l]}(m,n)$ is associated with $dA_{ij}^{[l]}$ by the relations:

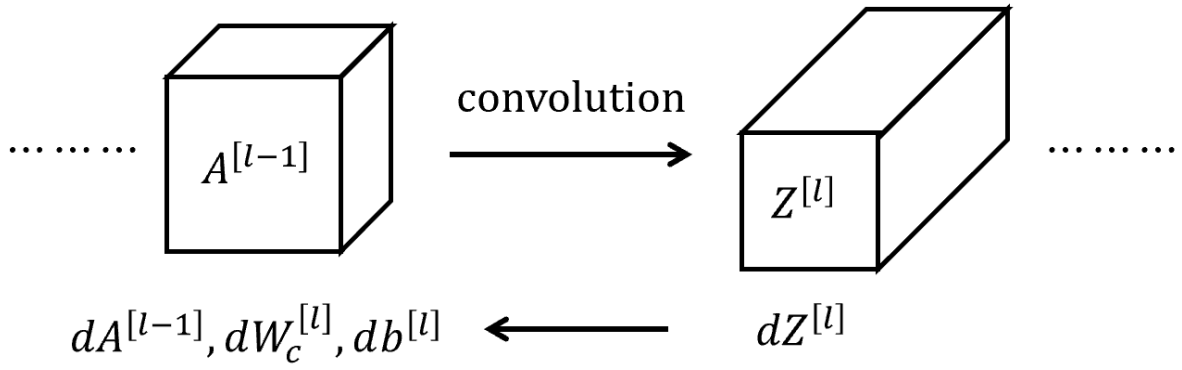
$$i = (h-1)s + m \qquad j = (w-1)s + n$$

By iterating over all $dA_{\text{slice},hw}^{[l]}(m,n)$, we add piece by piece to find each of the element $dA_{ij}^{[l]}$, and hence the full derivative matrix $dA^{[l]}$.

Note that the discussions above only dealt with one single channel, but generalisation to multiple channels should be straightforward.

Backward Pass for Convolutional Layers

> Objective: compute $dA^{[l-1]}$, $dW_c^{[l]}$, $db^{[l]}$ given $dZ^{[l]}$.



Note that $dA^{[l-1]}$ would be $dP^{[l-1]}$ for an architecture of multiple layers of convolution filers followed by pooling layers.

Let's consider each element of $dz_{hw}^{[l]}$.

Recall that $z_{hw}^{[l]} = \sum_{m=1}^{f} \sum_{n=1}^{f} w_c^{[l]}(m,n) \cdot A_{\text{slice},hw}^{[l-1]}(m,n) + b^{[l]}$.

So, each $dz_{hw}^{[l]}$ has a contribution to:

- $f \times f$ elements of $dw_c^{[l]}(m,n) += A_{\text{slice},hw}^{[l-1]}(m,n) \times dz_{hw}^{[l]}$
- 1 element of $db^{[l]} += 1 \times dz_{hw}^{[l]}$
- $f \times f$ elements of $dA_{\text{slice},hw}^{[l-1]}(m,n) += w_c^{[l]}(m,n) \times dz_{hw}^{[l]}$

Therefore, for the weights of the convolutional filter:

$$dW_c^{[l]} = \sum_h \sum_w A_{\text{slice},hw}^{[l-1]} * dZ_{hw}^{[l]}$$
$$db^{[l]} = \sum_h \sum_w dZ_{hw}^{[l]}$$

For the activations $dA^{[l-1]}$, again we need to sum up the element-wise contributions of $dA_{\text{slice},hw}^{[l-1]}(m,n)$ to $dA^{[l-1]}(i,j)$ with

$$i = (h-1)s + m \qquad j = (w-1)s + n$$

Implementation Notes

Suppose we have a training set with m samples and the data are stored as `numpy` arrays with the shapes:

- activations $A^{[l]}$ / $Z^{[l]}$: $m \times n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$
- weights of convolutional filter $W_c^{[l]}$: $f \times f \times n_C^{[l-1]} \times n_C^{[l]}$
- biases of convolutional filter $b^{[l]}$: $1 \times 1 \times 1 \times n_C^{[l]}$

To find the pixel value at h^{th} row, w^{th} column, c^{th} channel for the i^{th} sample at layer l, the associated sliding window from the previous layer can be tracked down using the slicing operation.

```
A_slice_prev[h, w, c] = A_prev[i_s, h_start:h_end, w_start:w_end, c]
```

where:

```
h_start = h * stride
w_start = w * stride
h_end = h_start + f
w_end = w_start + f
```

It is possible to use nested `for` loops to iterate over each of the sliding windows:

```
for i_s in range(m):
    for h in range(n_H):
        for w in range(n_W):
            for c in range(n_C):
                A_slice_prev[h, w, c] = ...
```

Forward Propagation

- Convolution: `Z[i_s, h, w, c] = np.sum(A_slice_prev[h, w, c] * W[:, :, :, c]) + b[:, : ,:, c]`
- Max Pooling: `P[i_s, h, w, c] = np.max(A_slice_prev[h, w, c])`
- Average Pooling: `P[i_s, h, w, c] = np.mean(A_slice_prev[h, w, c])`

Backward Propagation

- Convolution:

```
dA_prev[i_s, h_start:h_end, w_start:w_end, :] += W[:, :, :, c] * dZ[i_s, h, w, c]
dW[:, :, :, c] += A_slice_prev[h, w, c] * dZ[i_s, h, w, c]
db[:, :, : c] += dZ[i_s, h, w, c]
```

- Max Pooling:

```
M[h, w] = (A_slice_prev[h, w, c] == np.max(A_slice_prev[h, w, c]))  ## creating a mask
dA_prev[i_s, h_start:h_end, w_start:w_end, :] += M[h, w] * dP[i_s, h, w, c]
```

- Average Pooling:

```
dA_prev[i_s, h_start:h_end, w_start:w_end, :] += 1/(f*f) * dP[i_s, h, w, c] * np.ones(f, f)
```

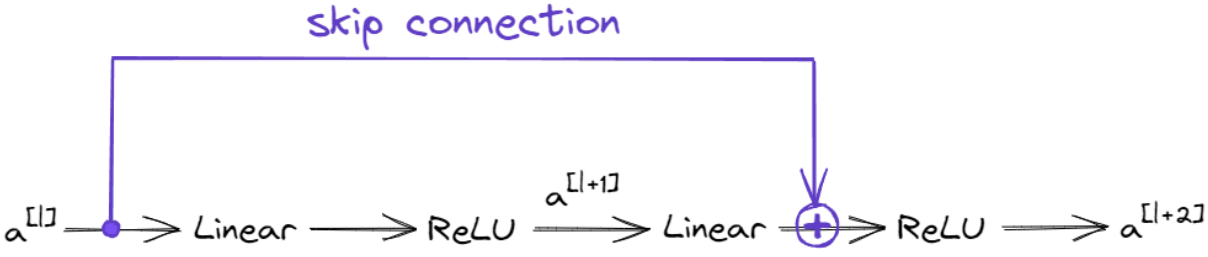
Course 3-2: CNN Architecture Case Studies

Residual Networks (ResNets)

Deeper networks tend to have better performance (can represent more complex functions, and also can learn features at many different levels of abstraction), but the network would suffer from problems of vanishing or exploding gradients as it goes deeper.

ResNets introduce **shortcuts**, or **skip connections**, across two or more layers. Stacking ResNet blocks on top of each other makes training very deep neural networks possible.

The diagram shows a skip connection being added between the l^{th} layer and the $(l+2)^{\text{th}}$ layer, which can be symbollically represented as: $a^{[l+2]} = g\left(z^{[l+2]} + a^{[l]}\right)$

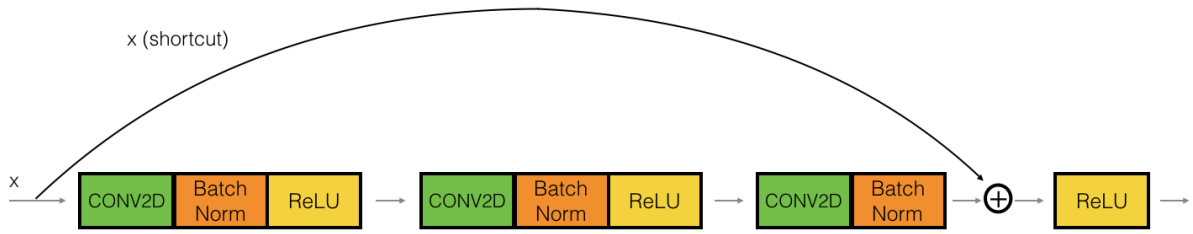


Why ResNets Work?

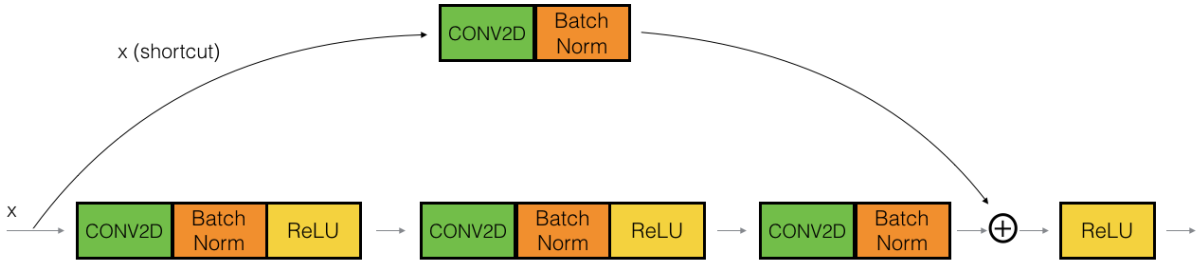
A residual block can easily learn an **identity function**. By tuning the weights $w^{[l+2]}$ and $b^{[l+2]}$ to zero, then the activation $z^{[l+2]} = w^{[l+2]}a^{[l+1]} + b^{[l+2]} = 0$, so $a^{[l+2]} = \text{ReLU}\left(a^{[l]}\right) = a^{[l]}$. This means that adding these two layers have little risk of harming the overall performance of the neural network.

Typical Residual Blocks

- **Identity block**: input activation has the same dimention as output activation



- **Convolutional block**: dimensions of input and output do not match up, so an additional convolutional layer is needed in the shortcut path to adjust the dimension of the previous activations before forwarding them to the upcoming layer



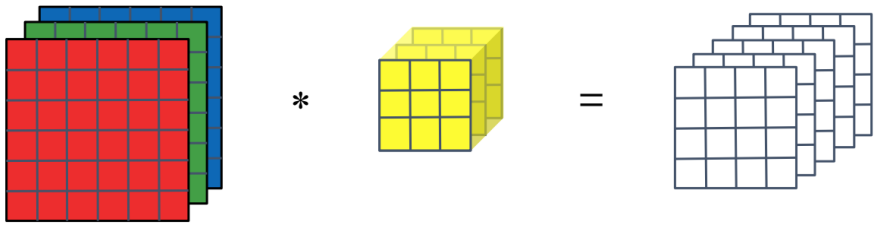
Depthwise Separable Convolutions

Traditional convolutions can be very resource intensive, and **depthwise separable convolution**s can reduce the number of trainable parameters and operations, and hence speed up the computations.

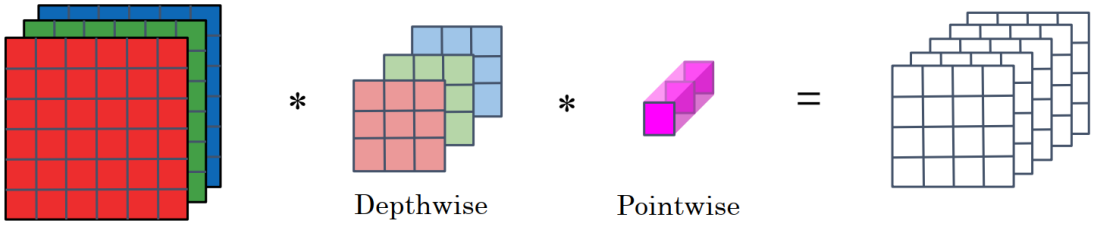
For normal convolution operation, suppose the input data has dimensions $n^{[l]} \times n^{[l]} \times n_C^{[l]}$ and the output data has dimensions $n^{[l+1]} \times n^{[l+1]} \times n_C^{[l+1]}$, and we are using $N = n_C^{[l+1]}$ filters of size $f \times f \times n_C^{[l]}$, then the total number of multiplication in this convolution operation is: $N \times {n^{[l+1]}}^2 \times f^2 \times n_C^{[l]}$.

Let's see how the computations can be greatly reduced by breaking down this single convolution operation into a depthwise convolution followed by a pointwise convolution.

Normal Convolution



Depthwise Separable Convolution



In depthwise step, instead of applying convolution to all the $n_C^{[l]}$ channels, the convolution is applied to a single channel at a time. So the filters will be of the size $f \times f \times 1$ and $n_C^{[l]}$ of such filters are required. The number of multiplications at depthwise step is: $n_C^{[l]} \times {n^{[l+1]}}^2 \times f^2$.

Next, in the pointwise step, a 1×1 convolution is applied on the $n_C^{[l]}$ channels. So the filter size for this operation is $1 \times 1 \times n_C^{[l]}$, and we would need $N = n_C^{[l+1]}$ such filters to match the dimension of output data. The number of multiplications at point-wise step is: $N \times n_C^{[l]} \times {n^{[l+1]}}^2$.

For the overall depthwise separable operation, the total number of multiplications: $n_C^{[l]} \times {n^{[l+1]}}^2 \times (f^2 + N)$.

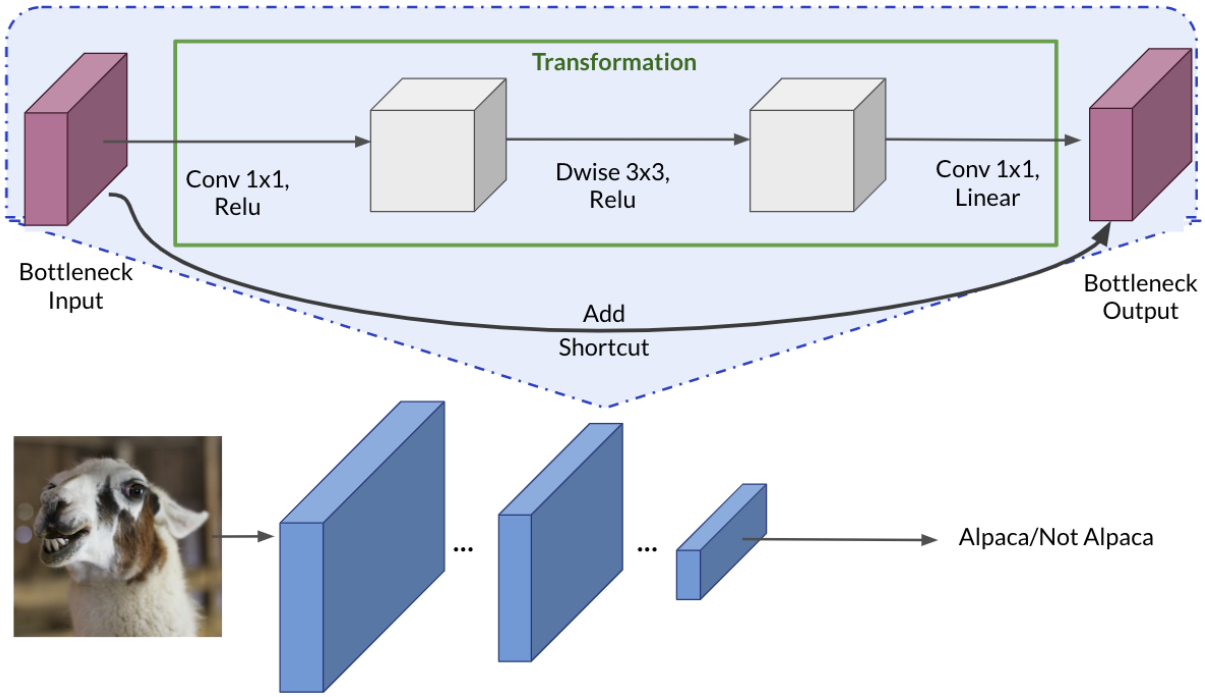
Comparing normal convolution operation with depthwise separable convolution, we find:

$$\frac{\#. \text{ of mul. of depthwise separable convolution}}{\#. \text{ of mul. of normal convolution}} = \frac{f^2+N}{N \times f^2} = \frac{1}{N} + \frac{1}{f^2}$$

Take $N = 512$ and $f = 5$ as an example, the ratio is found to be ~4.2%, so this depthwise seperable block performs over 20 times fewer multiplications as compared to a normal convolutional block. This suggests that we can deploy faster convolution neural network models without losing much of the accuracy.

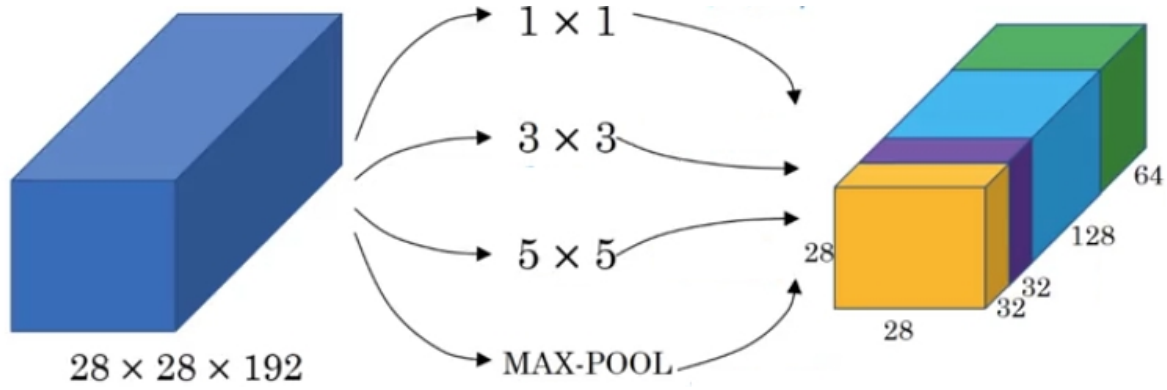
MobileNetV2 Architecture

The diagram below shows the architecture of MobileNetV2, which takes advantage of depthwise separable convolutions together with shortcut connections to speed up training and improve predictions. This allows MobileNetV2 to be run on mobile or other low-power applications with good efficiency.

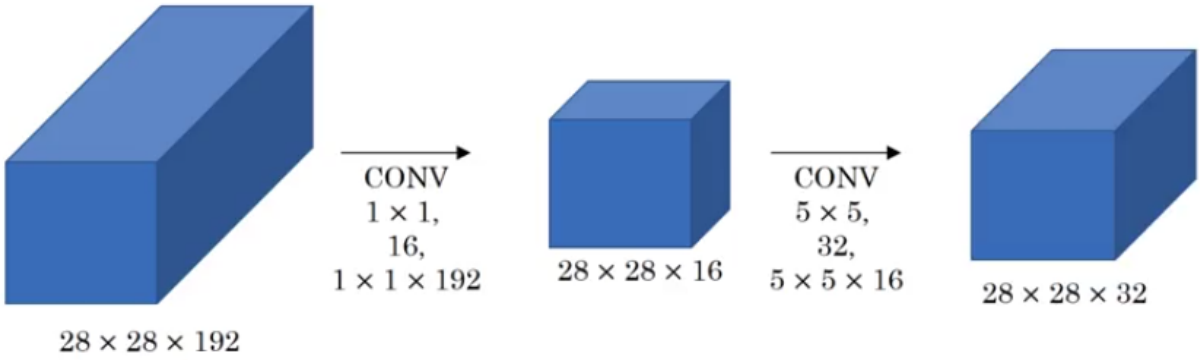


Inception Networks

We can apply different convolutions and pooling with filters of multiple sizes at the same layer, and concatenate them to give an output volume.



One problem with such inception block is its high computation cost. In order to save computations, we can shrink the number of channels by using 1×1 convolution filters. The following example shows how the number of computations is greatly reduced by the bottleneck layer of 1×1 convolutions.



Further Advices

Transfer Learning

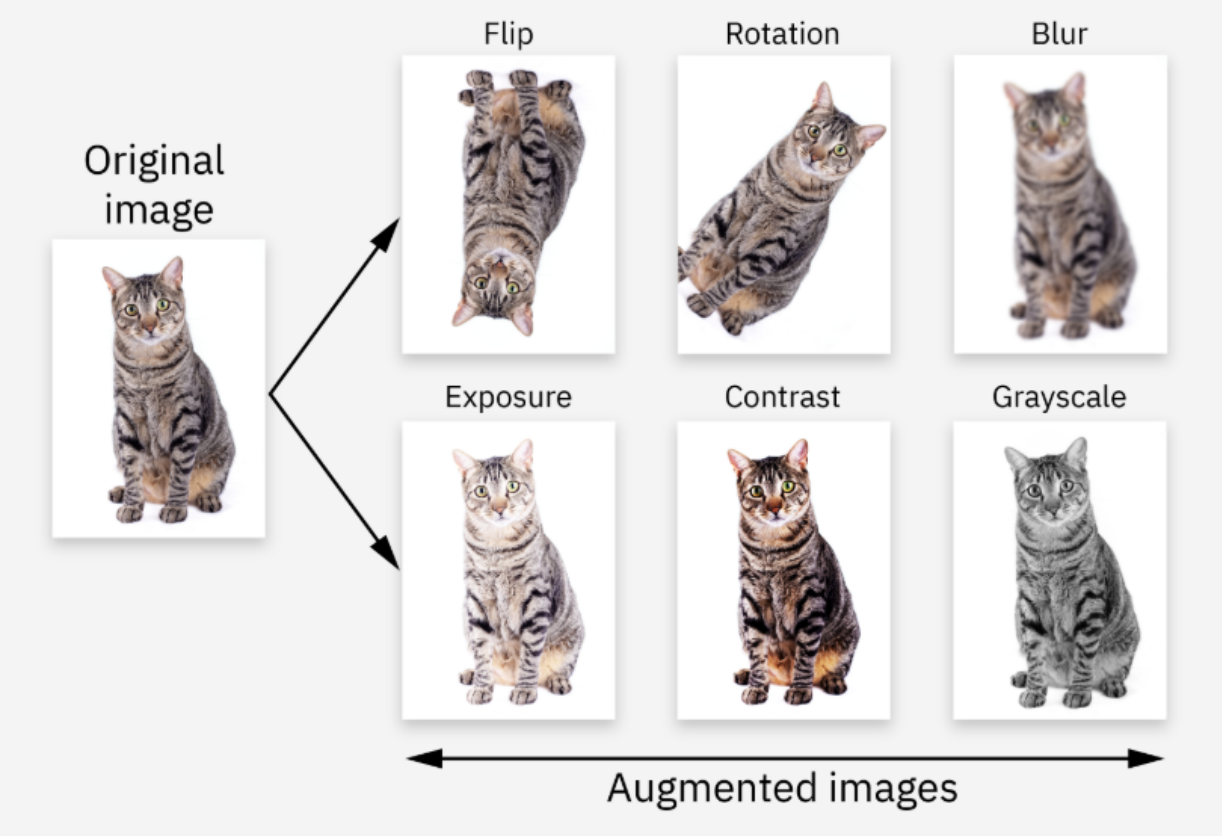
Many pre-trained models have been trained on very large datasets and have learned those weights with optimized hyperparameters.

For our own particular problem, instead of training a NN from scratch, we can use a specific NN architecture that has been trained by someone else. By replacing the output layer with a new one while keeping all the previous layers fixed, we only need to fine tune the last layer to fit the training examples. We can even compute the last activation for all training examples and save them to disk to reduce computations.

If we have enough data and computation power, instead of starting training a NN with random initialisations, we can initialise the weights with the parameters from pre-trained models and run optimisation algorithms from there. This, in general, can greatly reduce the amount of model training time.

Data Augmentation

Data augmentation is the process of artificially generating new data from existing data. This method helps us to have more training examples when we do not have enough data.



Common data augmentation methods used for computer vision tasks include:

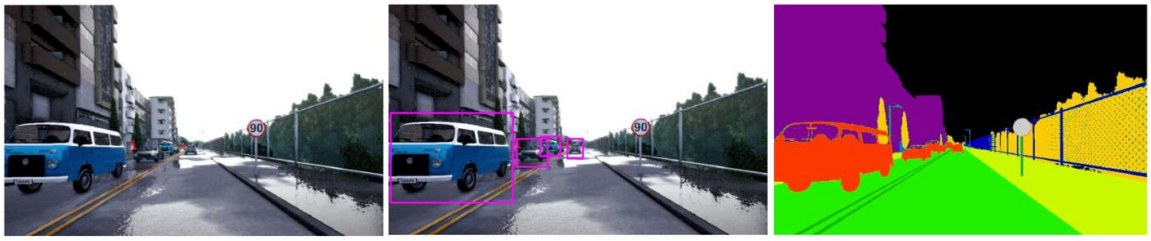
- geometric transformations (mirroring, random cropping, rotation, shearing, etc.)
- colour space transformations (add RGB distortions to the image, change contrast, change brightness, etc.)

It is also possible to add noise or randomly erase some part of the image to improve the model performance.

Course 3-3A: Object Detection

Typical Tasks in Computer Vision

- **Image Classification**: classify an image to a specific class (usually only one object in the whole image)
- **Classification with Localisation**: learn the class of the object, and generate a bounding box to give the location of the object in the image (usually only one object in the whole image)
- **Object Detection**: detect all the objects in the image and predict their classes and give their locations (usually more than one object from different classes in the image)
- **Semantic Segmentation**: label each pixel in the image with a category label



Input image | Object Detection | Semantic Segmentation

Object Detection

Defining the Outputs

The output vector y in the classification with localisation problem can be defined as:

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_w \\ b_h \\ C_1 \\ C_2 \\ \vdots \\ C_s \end{bmatrix}$$

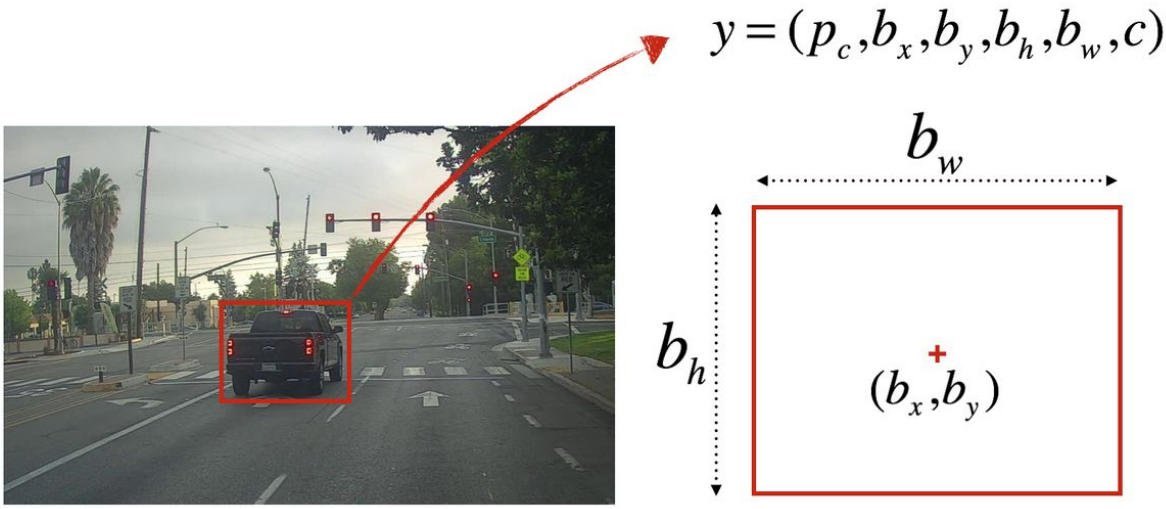
where

- P_c gives the probability if there is an object in the image
- b_x and b_y give the coordinates of the centre of the object
- b_w and b_h are the width and height of the object
- $C_1, C_2, \cdots, C_s$ are the probabilities that the object belongs to each of the s classes

The output y can also be defined to be:

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_w \\ b_h \\ c \end{bmatrix}$$

where c is the class index that represents one of the s classes to which the object belongs.

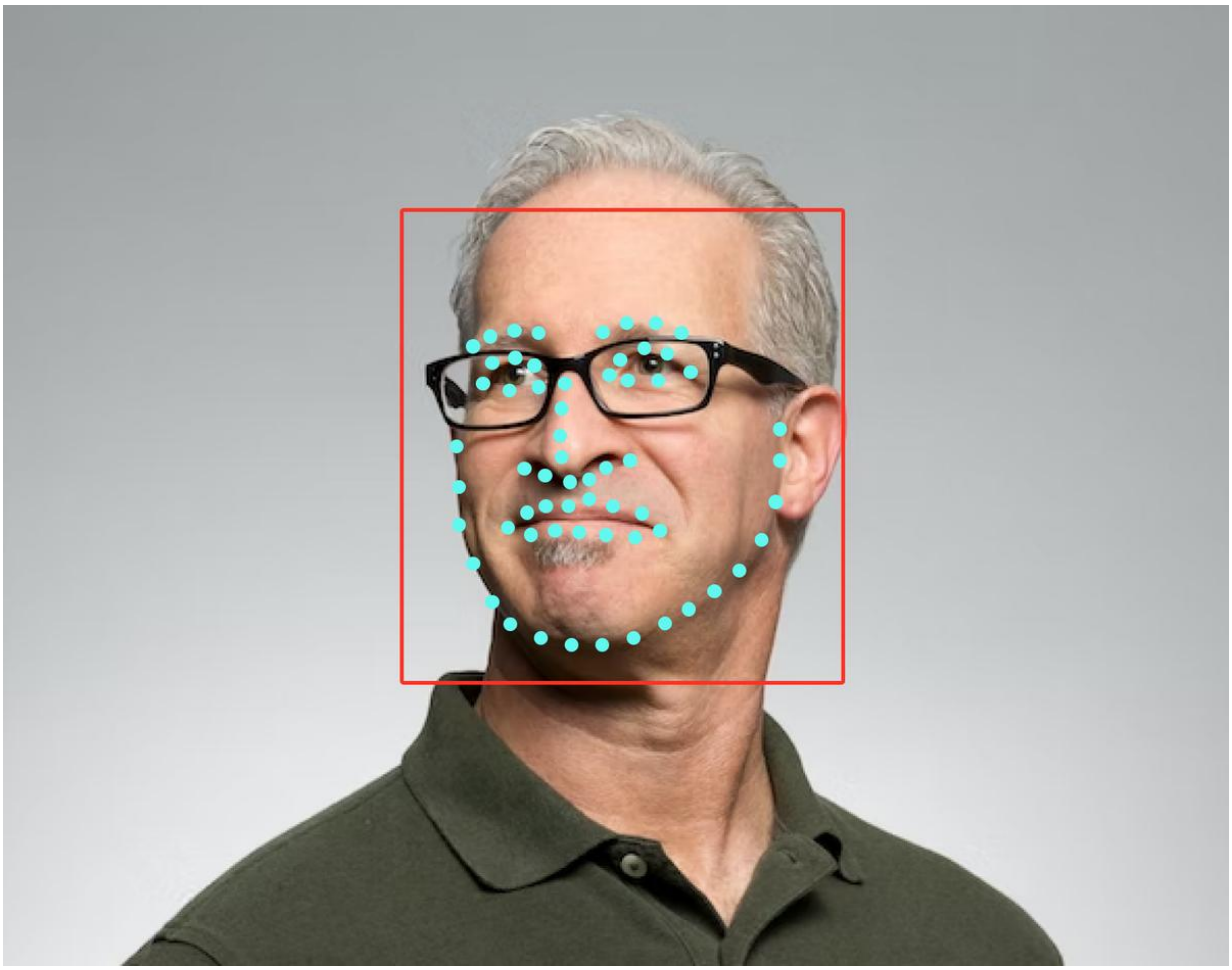


$p_c = 1$: confidence of an object being present in the bounding box

$c = 3$: class of the object being detected (here 3 for “car”)

Landmarks

In computer vision problems like face recognition problems, we might also want to output some points on the face like corners of the eyes, corners of he mouth, corners of the nose and so on, which makes it possible to predict the facial expressions of that person. Another example is when we get the skeleton of a person, outputting the positions of the hands, elbows, shoulders, knees, feet and so on could be helpful in predicting what the person is doing. This is called **landmark detection**.



Instead of using four numbers to give the location and the size of the object of interest, a collection of landmarks are needed in the labelled data.

$$y = \begin{bmatrix} P_c \\ l_{1,x} \\ l_{1,y} \\ l_{2,x} \\ l_{2,y} \\ \vdots \\ l_{s,x} \\ l_{s,y} \end{bmatrix}$$

where $(l_{i,x}, l_{i,y})$ are the coordinates of the i^{th} landmark which can be the left corner of the left eye, or the right corner of the nose, etc.

Choosing the Loss Function

There are two primary tasks in an object detection problem: **classification** (identifying the class of the object) and **localisation** (specifying the object's position and size). In practice, we use a combination of loss functions.

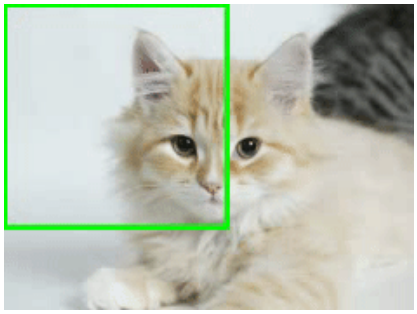
In the course, Andrew's recommendations are:

- logistic regression for P_c and log-likelihood loss for the classes (standard loss function for most classification tasks)
- mean squared error for the bounding boxes

Sliding Window Technique

For object detection problems, there could be many objects scattered at multiple places across an image, but we can build upon what we had learned earlier in this course and use the sliding window technique.

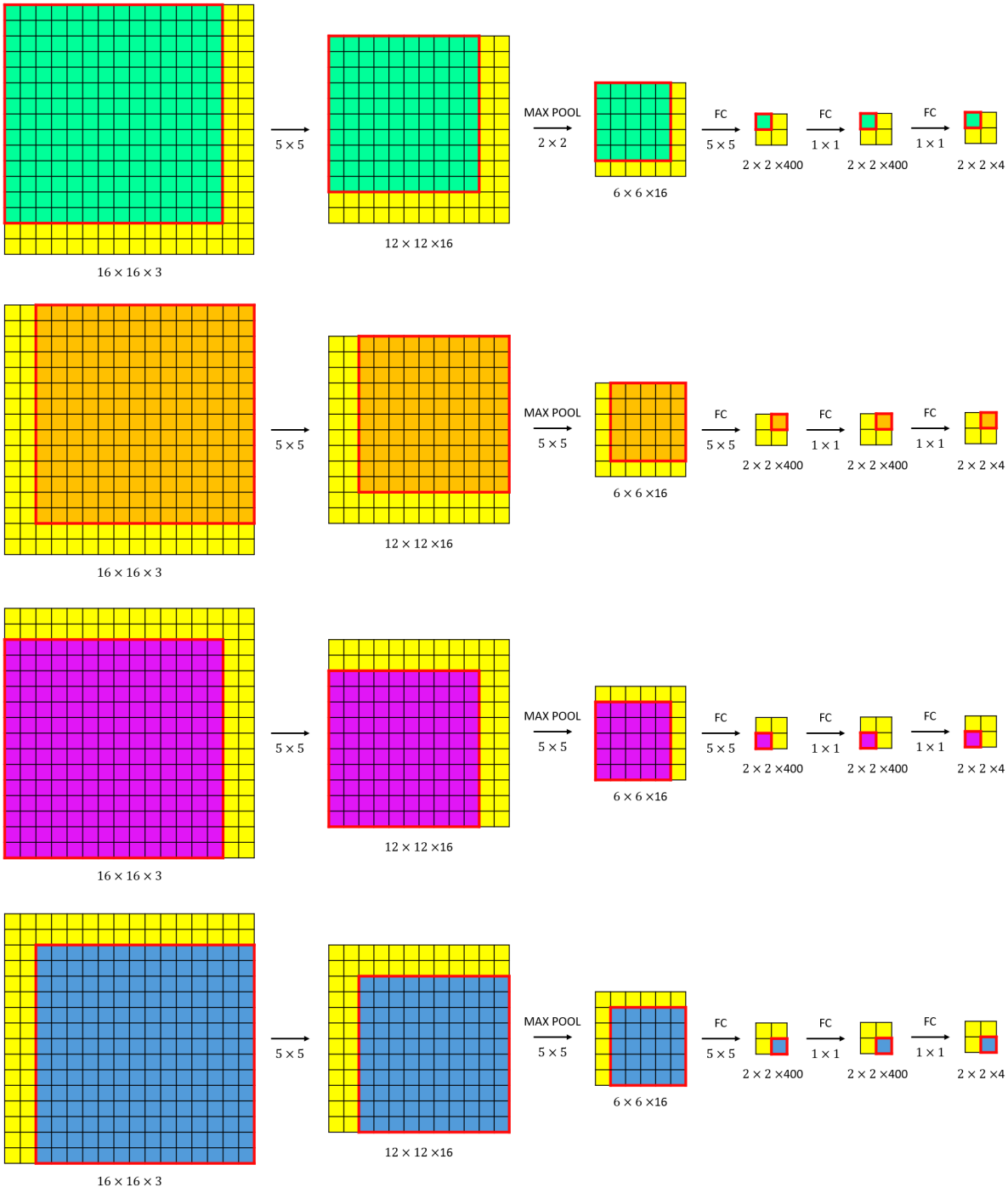
Recall that we have learned how to train a CNN for the image classification problems, that is to identity the only object in one image. For object detection, we can crop specific windows of the images (with varying sizes and varying ratios) and forward the cropped portion into a CNN and predict the corresponding class for each window.



The disadvantage of this method is its high computational cost. It turns out that the sliding windows can be implemented more efficiently using convolutional networks.

Sliding Window with Convolutional Implementation

The illustration below shows how a convolutional network takes a $16 \times 16 \times 3$ image and produces a $2 \times 2 \times 4$ output. Each $1 \times 1 \times 4$ slice in the output makes a prediction about the probability that each corresponding sliding window (as shown in different colours) belongs to each one of the 4 classes. Similar architectures can be applied for bigger images and more classes.



The sliding window approach treats each window as an independent image patch separately, but neighbouring sliding windows with large fractions of overlapping may share a lot of repeated computations. Therefore, processing the entire image as a whole with a CNN, which shares and reuses the feature map for multiple regions, can significantly eliminate redundant computations and hence reduce computational cost.

YOLO

YOLO (*You Only Look Once*) is a popular algorithm for object detection problems because of its high accuracy and also its ability to run in real time. This algorithm was named YOLO as it requires only one forward propagation pass through the network to make predictions, so in this sense it only looks once at the image. After non-max suppression, it then outputs recognized objects together with the bounding boxes.

Bounding Box Prediction

By dividing the input image into a smaller grid of size $G \times G$, we can perform a simple object localization for each grid cell where the network outputs the class probabilities and the bounding boxes of the main object in that grid cell. For each grid cell, the network detects the object for which its centre belongs to that grid cell, even if the object may span into neighbouring grid cells.

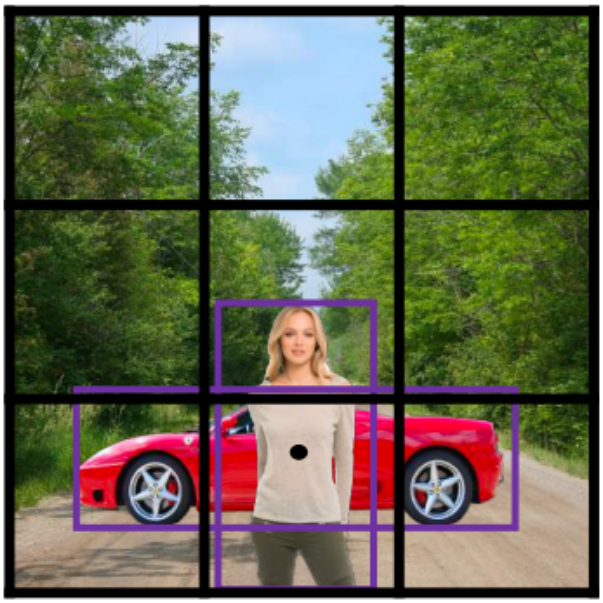
For each grid cell, if we define the target label as mentioned earlier as

$$y = \begin{bmatrix} P_c \\ b_x \\ b_y \\ b_w \\ b_h \\ C_1 \\ C_2 \\ \vdots \\ C_s \end{bmatrix}$$

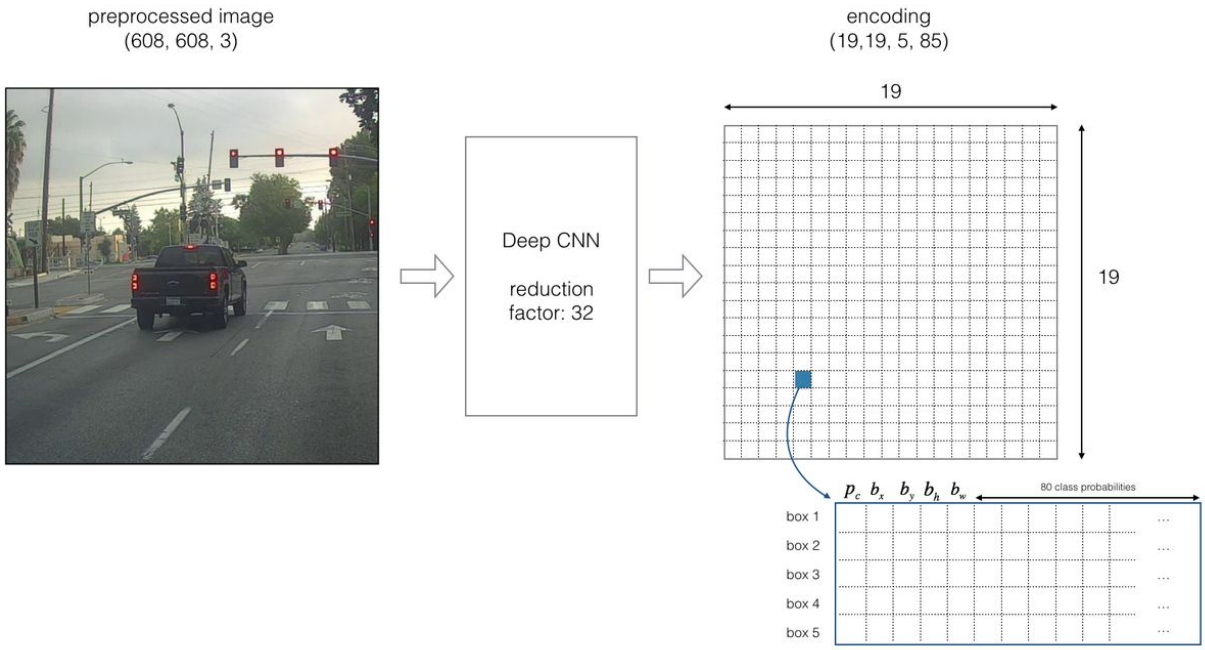
then the dimension of the output layer is $G \times G \times (s + 5)$.

Anchor Boxes

Objects from different classes usually have different shapes. For example, we may expect the bounding box for a pedestrian to be tall and thin, while the bounding box for a car to be relatively wider. To represent different objects in the training data, we choose reasonable height/width ratios for different classes. Such bounding boxes with predefined reference shapes are called **anchor boxes**.



Suppose we have n_A anchor boxes, then the dimension of the output tensor of the last layer is $G \times G \times n_A \times (s + 5)$.



Class Scores

We can further extract a probability that the box contains a certain class by computing the element-wise product $P_c \times C_i$. After calculating the score for all s classes in one anchor box, we may identify the maximum score and assign to this anchor box this class score and the corresponding class.

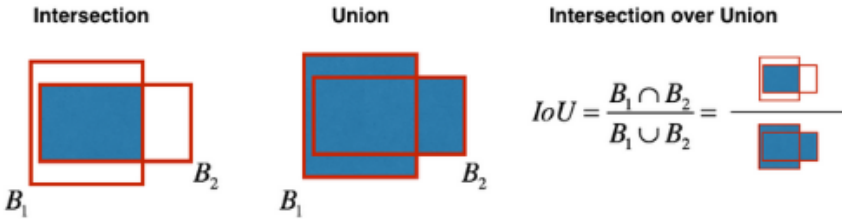
Non-Max Suppression

One problem with YOLO at this stage is that it detects the same object multiple times. The model would output $G \times G \times n_A$ boxes, which are way too many boxes. We need to reduce the algorithm's output to a much smaller number of detected objects. To do so, we can use non-max suppression to make sure that YOLO detect each object just once. This involves the following steps:

- choose a threshold for the class scores and get rid of the boxes with a low class score
- select only one box when several boxes overlap with each other but detect the same object

To determine which boxes detect the same object, one measure that can be used is the **intersection over union**. The *IoU* between two bounding boxes B_1 and B_2 is defined as:

$$IoU = \frac{B_1 \cap B_2}{B_1 \cup B_2}$$

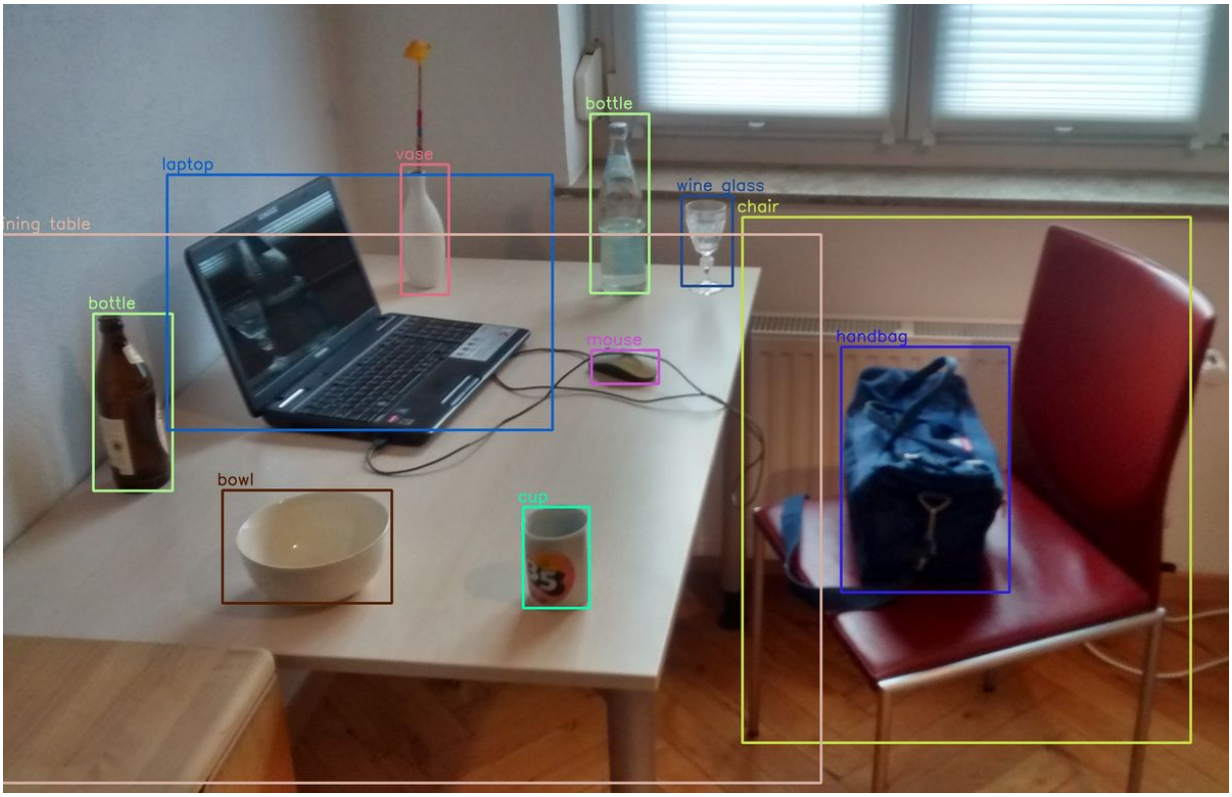


While there are too many remaining boxes, we pick the box with the largest P_c and output that as a prediction. At the mean time, we search for and remove any remaining box with the same output but with an *IoU* that is greater than a certain threshold.

If there are s classes to be detected, we should run non-max suppression s times, once for each output class.

Visualising YOLO

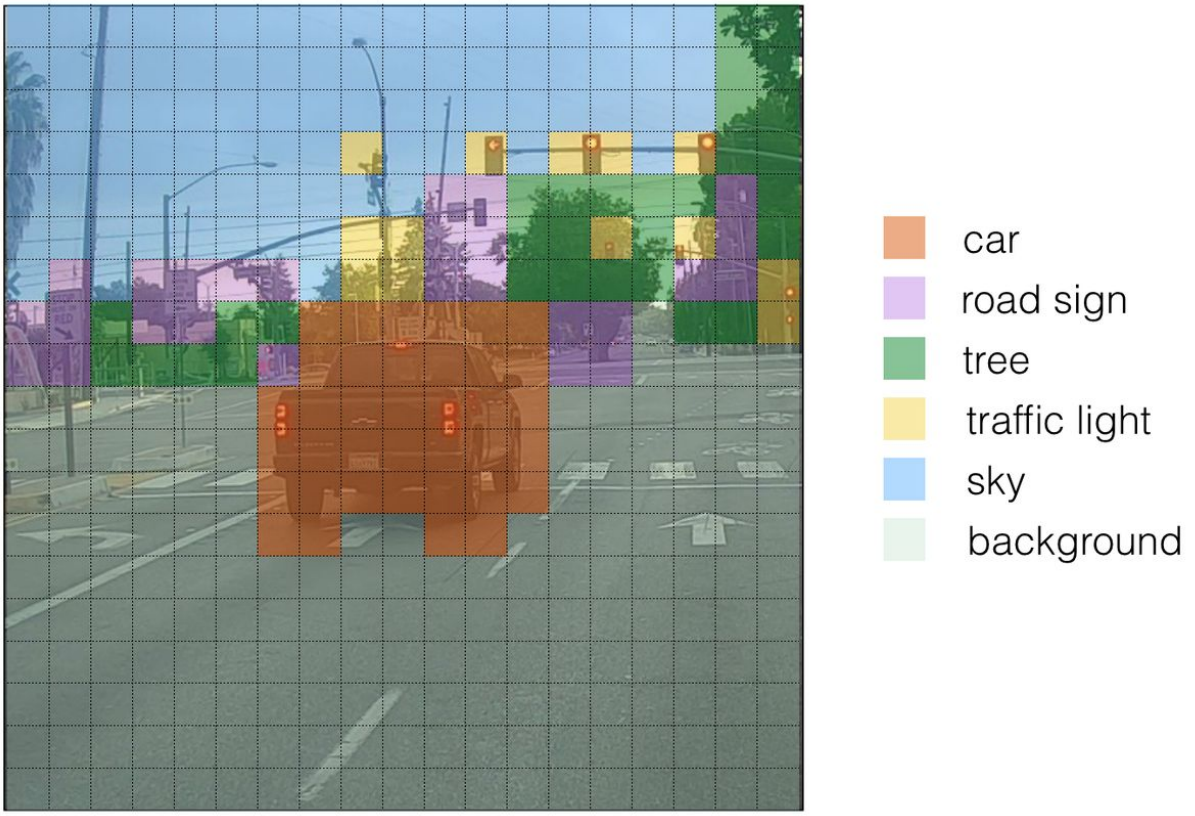
One way to visualize YOLO's output is to plot the bounding boxes that it predicts. Doing this results in a picture like this:



Another way to visualise what YOLO predicts on an image is to do the following:

- for each grid cell, find the highest probability score (take a maximum score across the s classes, and one maximum for each of the n_A anchor boxes)
- colour that grid cell according to what object that grid cell considers the most likely

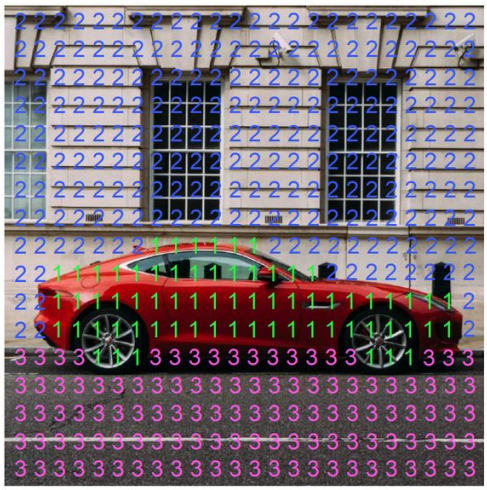
Doing this results in a picture that looks like this:



Course 3-3B: Semantic Segmentation

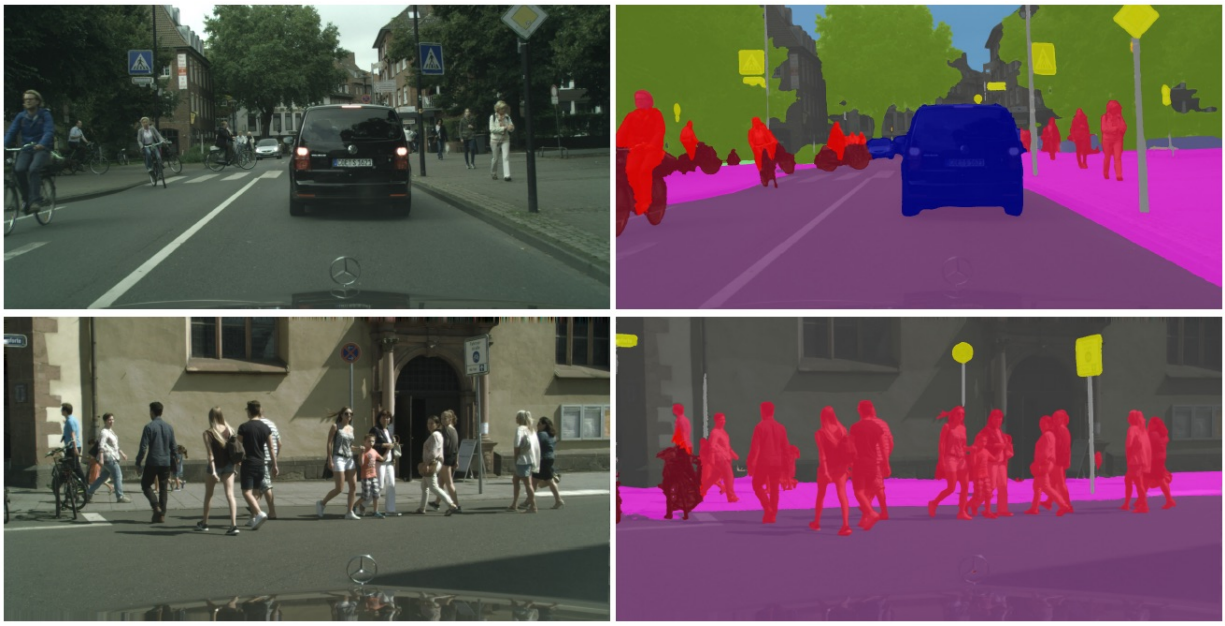
Semantic Segmentation: Overview

Semantic image segmentation is the task of labelling each pixel of an image into a predefined set of classes. The output assigns a semantic class label to each pixel, so a segmented map can be drawn such that regions of the same class labelled with the same colour.



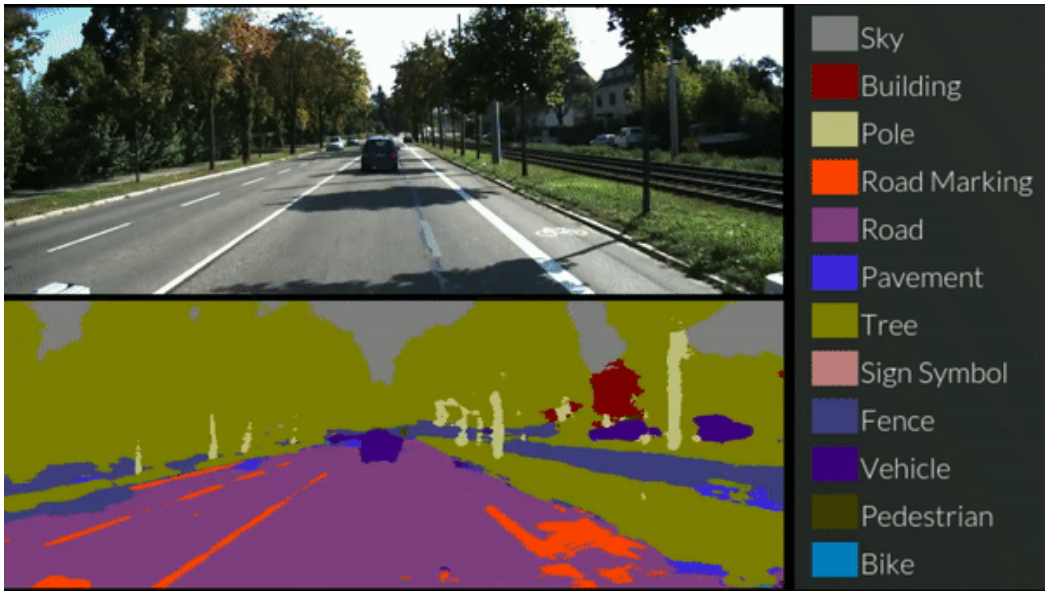
1. Car
2. Building
3. Road

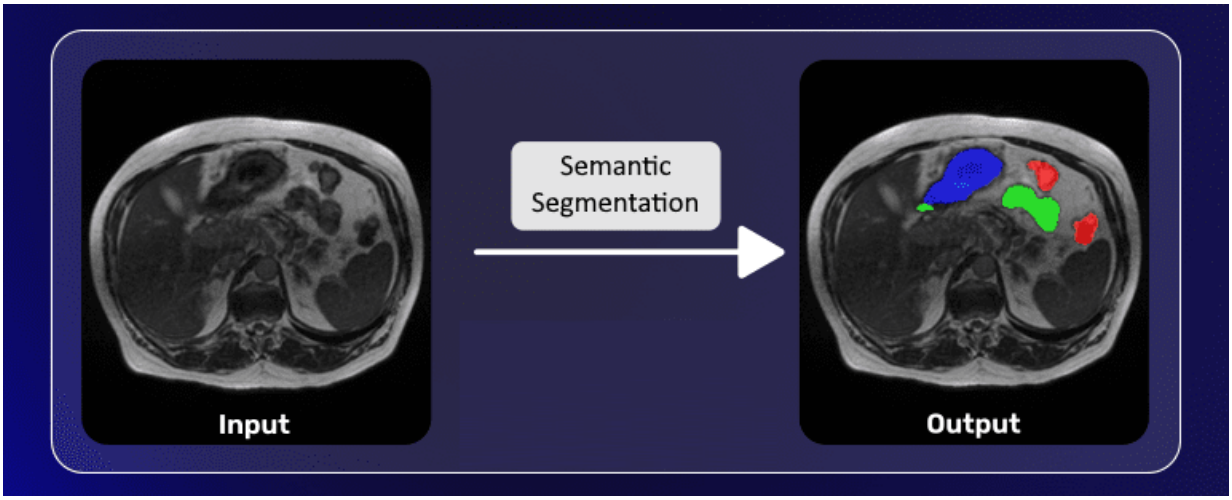
Segmentation Map

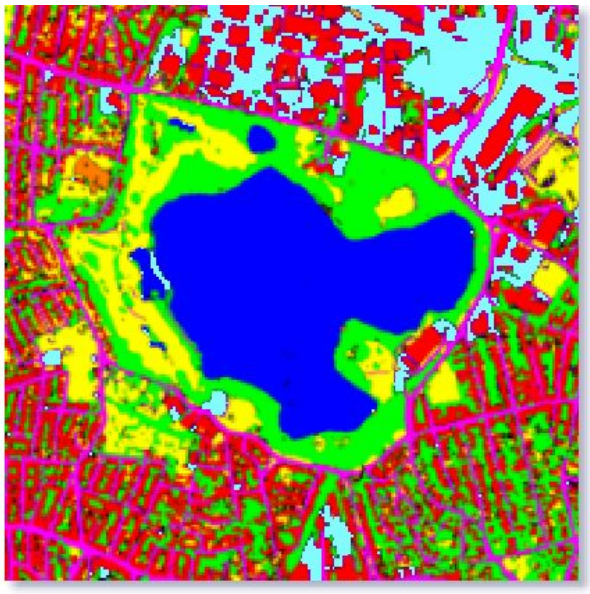


Applications of semantic segmentations include

- Autonomous vehicles: help car distinguish roads, lanes, pedestrians and obstacles for safer navigation
- Medical imaging: distinguish organs, tumours and tissues with high precision for diagnostics
- Satellite imagery: map land use, model cities, analyse urban development, monitor water bodies, etc.
- Augmented reality and photography: enable live background replacement, portrait modes and advanced filters



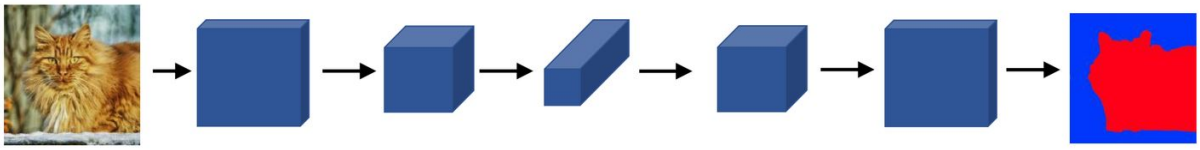




Fully Convolutional Networks (FCNs)

Like other computer vision tasks, we use a CNN for semantic segmentation. However, unlike image classification problems, where the size of the input image gets **downsampled** through a series of strided convolutional and pooling layers and is finally fed into fully-connected layers for classification purposes, semantic segmentation problems requires the output have the same resolution as the input image.

To retain the spatial information that is lost during the downsampling phase, we replace the fully-connected layers in the network by a series of **upsampling** layers followed by more convolutional layers to reproduce higher resolution feature maps. This architecture is called a **Fully Convolutional Network (FCN)**.



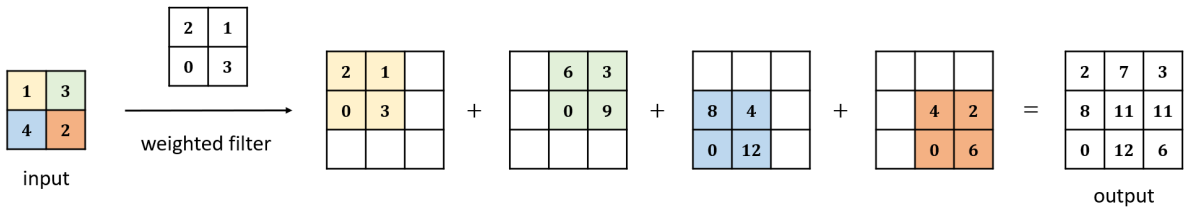
The reason behind this general architecture is the following. A typical CNN starts with a high resolution image so it is impractical to connect each neuron to all other neurons. Hence, the initial layers in a CNN can only capture information about smaller regions of the image and learn low-level features like lines, edges and colours. As the feature map is passed through more layers, the size of the image keeps on decreasing and the number of the channels keeps on increasing. Despite the loss of spatial information, the deeper layers become able to learn high-level features like faces and objects. These high-level information about the input image is contained in its various channels.

Now that we have obtained this low-resolution tensor, we have to increase its resolution up to the original image to achieve the task of semantic segmentation. During the upsampling phase, the resolution of the image gets restored while the number of the channels in the feature maps decreases.

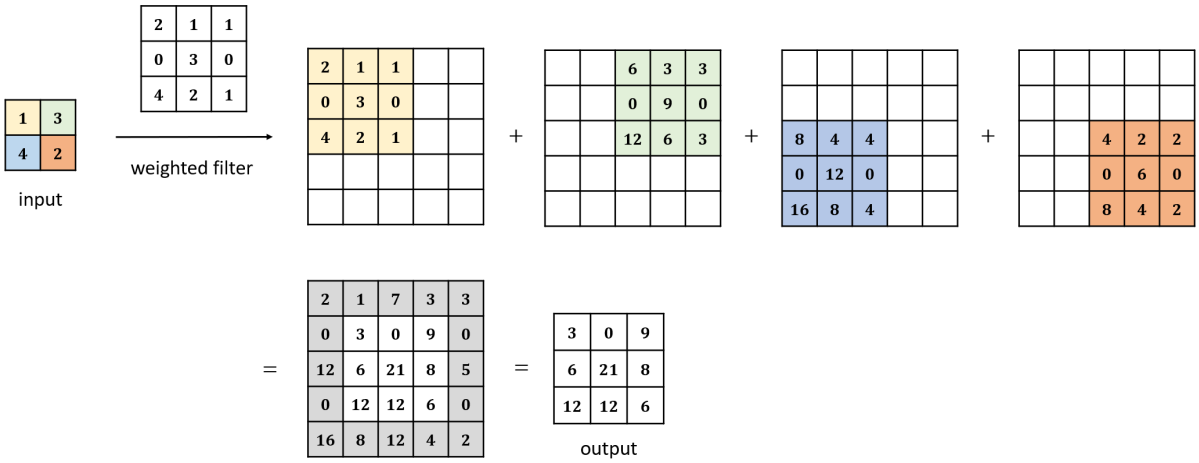
Transpose Convolution

Transpose convolution, also called fractionally-strided convolution, is a type of CNN layer that is useful for tasks that involve upsampling. Instead of sliding the filter over the input, a transposed convolutional layer slides the input over the filter. The element-wise multiplication and summations are performed in a similar way.

The example below illustrates how the transposed convolution with a 2×2 filter is computed for a 2×2 input tensor with a stride of 1 and no padding.



The next example illustrates how the transposed convolution is computed with stride $s = 2$ and padding $p = 1$.



In general, if an input feature map of size $n_h \times n_w$ is passed through a weighted filter of size $f \times f$ with a stride of s and padding of p, then the output of the transposed convolutional layer will be:

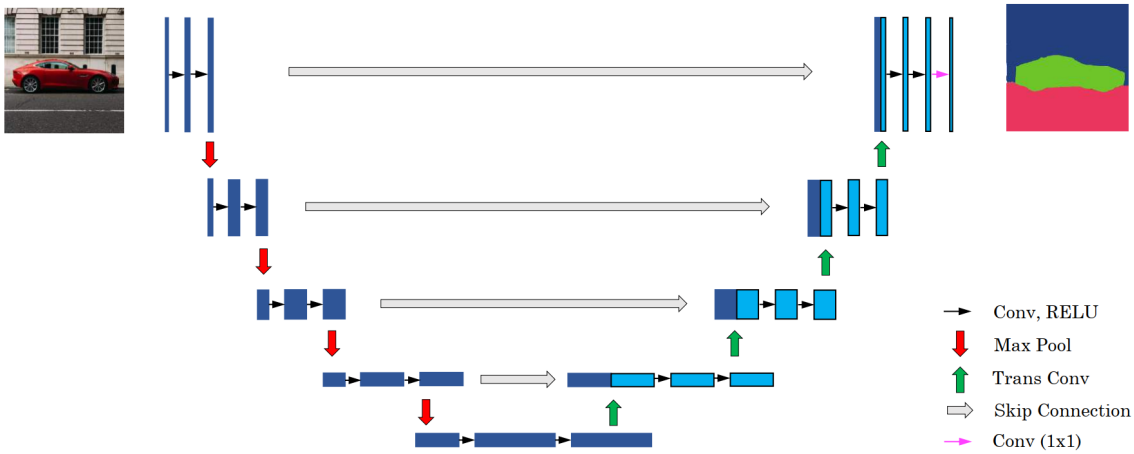
$$((n_h - 1) \times s + f - 2p) \times ((n_w - 1) \times s + f - 2p)$$

This could result in an output that is larger than the input, and hence increase the spatial dimensions of the feature maps.

U-Net

U-Net, named for its U-shape, was originally created for tumour detection, but has now become a popular choice for many other semantic segmentation tasks.

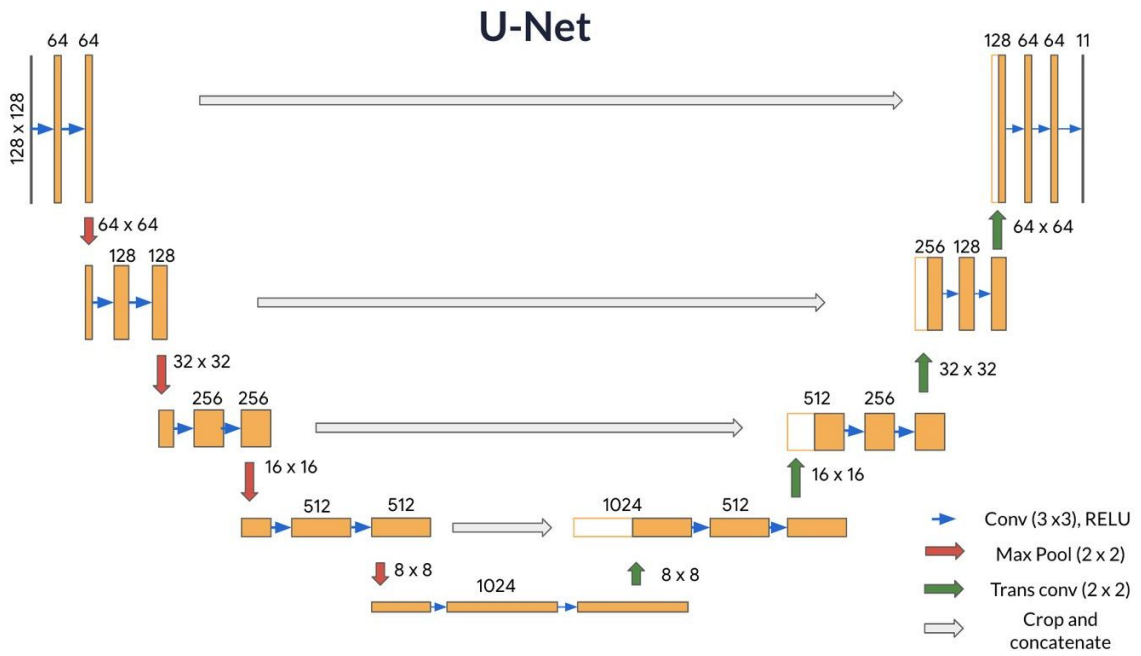
U-Net improves on the FCN, using a somewhat similar design, but differing in some important ways. It uses a matching number of convolutions for downsampling and transposed convolutions for upsampling. It also adds **skip connections**, to retain information that would otherwise become lost during encoding. Skip connections send information to every upsampling layer in the decoder from the corresponding downsampling layer in the encoder, capturing finer information while also keeping computation low. These help prevent information loss, as well as model overfitting.



[Ronneberger et al., 2015, U-Net: Convolutional Networks for Biomedical Image Segmentation] Andrew Ng

U-Net: Model Details

In the programming assignment of the course, we got to build our own U-Net for image segmentation. The architecture of this particular U-Net is shown below.



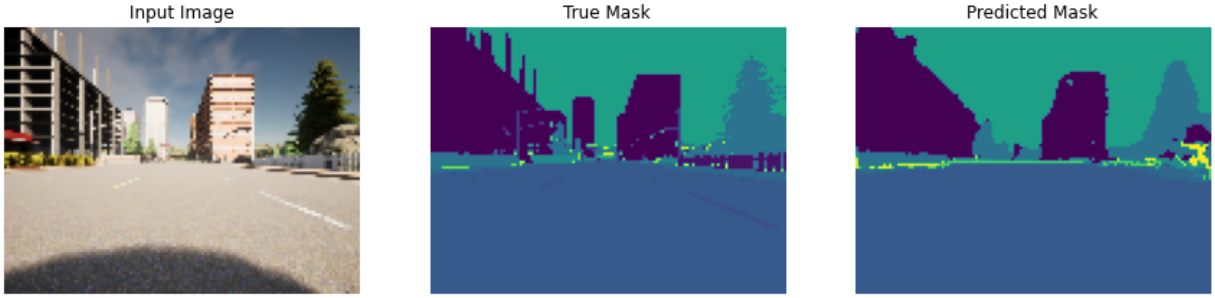
- **Contracting path** (Encoder containing downsampling steps): The contracting path follows a regular CNN architecture to downsample the image and extract its features. In detail, it consists of the repeated application of two 3×3 same padding convolutions, each followed by a ReLU unit and a 2×2 max pooling operation with stride 2 for downsampling. At each downsampling step, the number of feature channels is doubled.
- **Crop function**: This step crops the image from the contracting path and concatenates it to the current image on the expanding path to create a skip connection.
- **Expanding path** (Decoder containing upsampling steps): The expanding path performs the opposite operation of the contracting path, growing the image back to its original size, while shrinking the channels gradually. In detail, each step in the expanding path upsamples the feature map, followed by a 2×2 transposed convolution. This transposed convolution halves the number of feature channels, while growing the height and width of the image. Next is a concatenation with the correspondingly cropped feature map from the contracting path, and two 3×3 convolutions, each followed by a ReLU.
- **Final Feature Mapping Block**: In the final layer, a 1×1 convolution is used to map each 64-component feature vector to the desired number of classes. By choosing an appropriate number of 1×1 filters, the channel dimensions can be reduced to have one layer per class.

The U-Net network has 23 convolutional layers in total.

Experimental Results

The U-Net model for semantic image segmentation is implemented with **sparse categorical cross entropy** for pixelwise multiclass prediction and trained on on the **CARLA** self-driving car dataset.

Although the model was only trained for 40 epochs due to computational constraints for the assignment, we get some pretty amazing results.



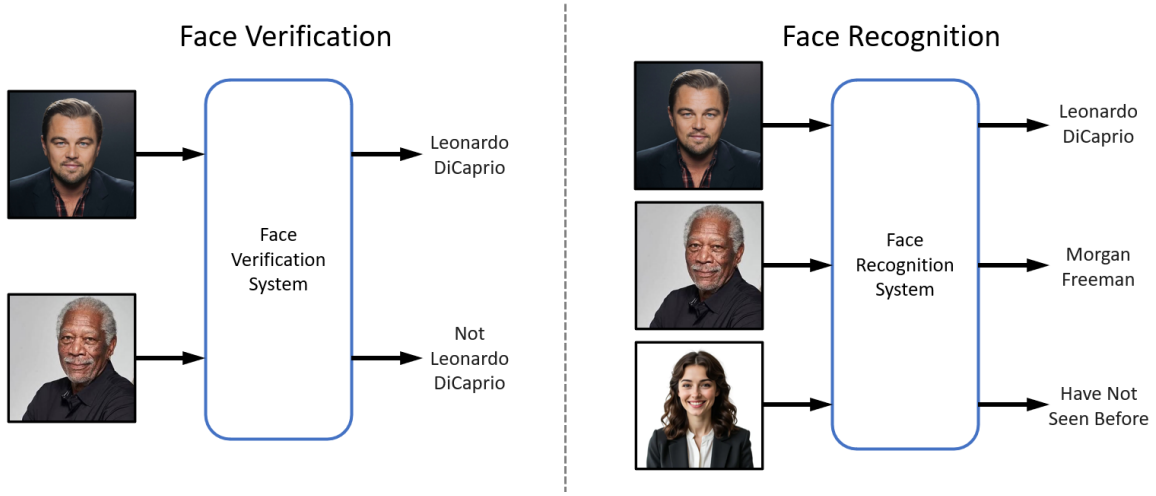
Course 3-4: Face Recognition & Neural Style Transfer

Face Recognition

Face recognition technology uses AI to analyze unique facial features, creating a mathematical encoding to identify or verify a person, common in access control systems like contactless entrance guards and phone unlocking (Face ID).

Face recognition problems commonly fall into one of the two categories:

- **Face Verification**
  - Input: image, name/ID
  - Output: whether the input image corresponds to the claimed person (1-to-1 matching problem)
- **Face Recognition**
  - Input: image
  - Output: whether the input image corresponds to any one of the registered persons in a database (1-to-K matching problem)



One-shot Learning

One of the difficulties in this approach is that we might only have one example image for each person to train the model. This is called **one-shot learning**.

Also, the model has to be retrained each time a new person is added to the database, and in test time, we need to compare an input image to all the instances in the database.

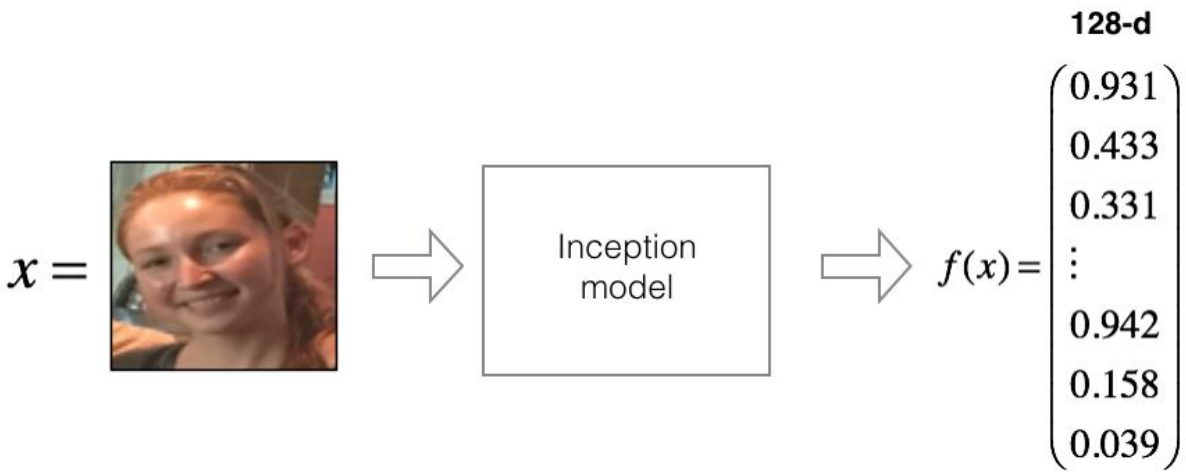
One possible solution for the one-shot learning problem is to learn a **similarity function**:

$$d(\text{img1}, \text{img2}) = \text{degree of difference between the input images}$$

- If $d(\text{img1}, \text{img2})$ is less than some threshold value, then we claim the two images show the same person.
- If $d(\text{img1}, \text{img2})$ is greater than the threshold, then the two images show two different persons.

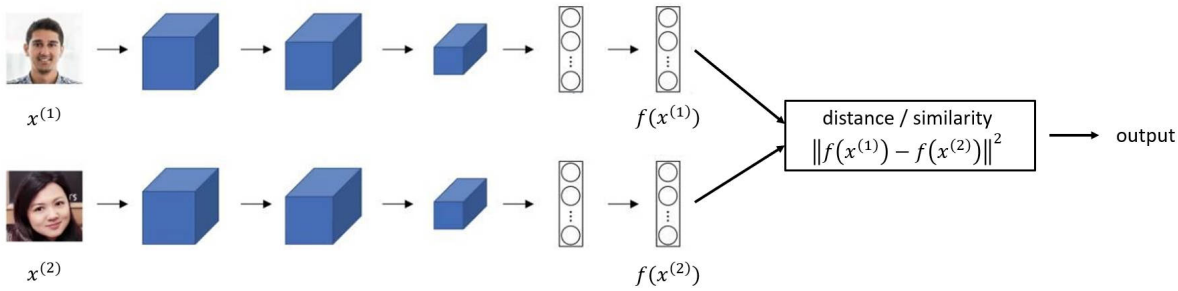
Siamese Network

To learn a similarity function, we use a **Siamese Network**, which takes a person's image x as the input and outputs an **encoding** $f(x)$. In the course's programming assignment, we used a 128-dimensional encoding in the output layer, i.e., the output is a vector of 128 components.



The goal of training is to learn parameters so that

- If $x^{(i)}$ and $x^{(j)}$ are the same person, then $\|f(x^{(i)}) - f(x^{(j)})\|^2$ is small.
- If $x^{(i)}$ and $x^{(j)}$ are different persons, then $\|f(x^{(i)}) - f(x^{(j)})\|^2$ is large.



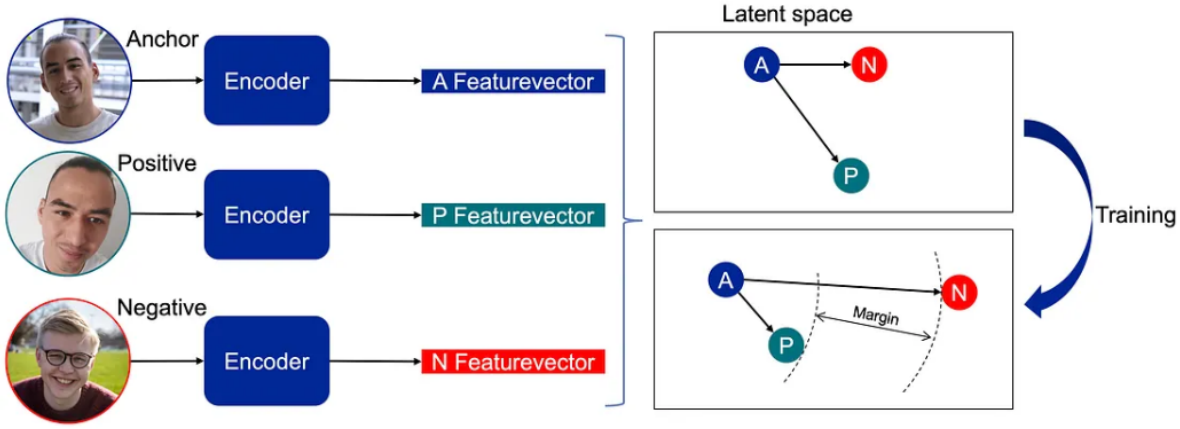
Triplet Loss

The **triplet loss** is an effective loss function for training a neural network to learn an encoding of a face image. In particular, training of the network use triplets of images (A, P, N), that is:

- **Anchor** A: an image of a person
- **Positive** P: an image of the same person as the Anchor
- **Negative** N: an image of a different person

The network will try to learn parameters that pushes the encodings of A and P closer together while pulling the encodings of A and N further apart. For even better differentiation, we would like to make sure that the Anchor image A is closer to the Positive image P than to the Negative image N by at least some margin α.

$$d(A, P) + \alpha < d(A, N)$$



If we measure the similarity with an L_2 distance, then this becomes

$$\|f(A) - f(P)\|^2 + \alpha < \|f(A) - f(N)\|^2$$

With a training dataset of many triplets, we would like to minimise the following triplet cost:

$$\mathcal{J} = \sum_{i=1}^{m} \max\left(\left[\|f(A^{(i)}) - f(P^{(i)})\|^2 - \|f(A^{(i)}) - f(N^{(i)})\|^2 + \alpha\right], 0\right)$$

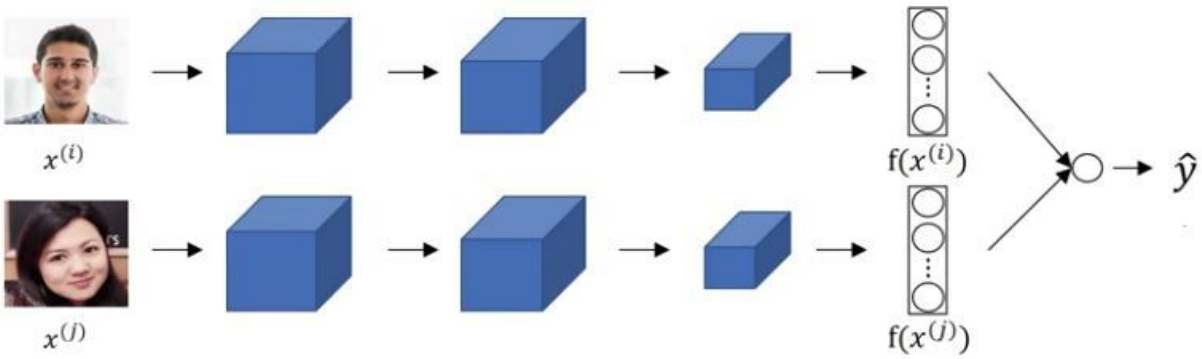
where $(A^{(i)}, P^{(i)}, N^{(i)})$ denotes the i-th training example.

Binary Classification

An alternative to the triplet loss approach is to formulate the face recognition problem as a binary classification problem. The CNN computes the encodings of two input images $x^{(i)}$ and $x^{(j)}$, and use a logistic regression unit to produce an output:

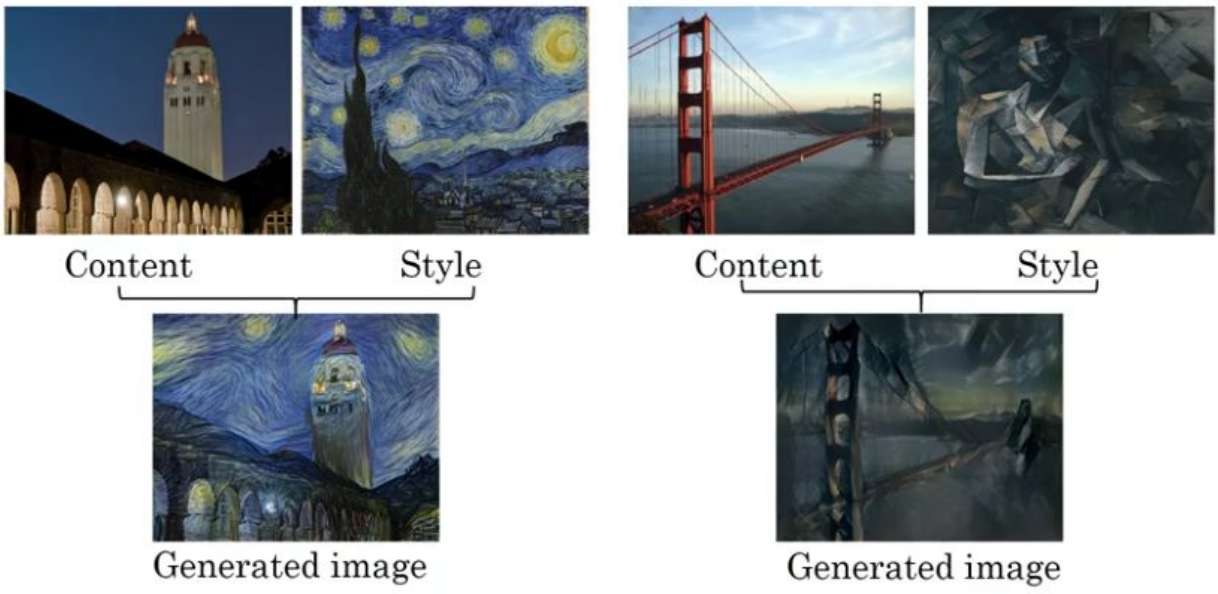
$$\hat{y} = \sigma\left(\sum_{k=1}^{m}\left(w_i|f(x^{(i)})_k - f(x^{(j)})_k| + b\right)\right)$$

- $\hat{y} = 1$ if the two images are of the same person
- $\hat{y} = 0$ if the two images are of the same person



Neural Style Transfer

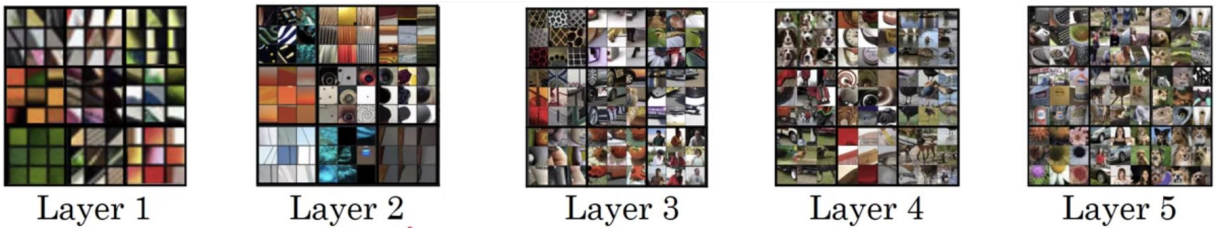
The goal of **Neural Style Transfer** (NST) is to merge two images, a **content image** (C) and a **style image** (S), to create a **generated image** (G) such that the generated image G combines the content of the image C with the style of image S.



This is implemented by extracting the content statistics of the content image C and the style statistics of the style image S using a deep convolutional network, and then optimizing the output image G to match these statistics.

Choosing the Layers of the Model

Starting from the network's input layer, the first few layer activations represent low-level features like edges, corner and simple textures. As we step through the network, the deeper layers detect higher-level features like parts of specific objects and more complex patterns.



Nerual Style Transfer uses *intermediate layers* of a pre-trained image classification network such as the VGG19 network, so that the content representation contains information about the image's structures and objects while discarding specific details that are not relevant to the main objective. Choosing intermediate layers allows the NST algorithm to control the balance between preserving the content image's structure and applying the style image's aesthetic.

Cost Functions for NST

The cost function that we use for the NST algorithm contains two terms:

- a **content cost function** $J_{\text{content}}(C, G)$ that measures the similiarity between the contents of C and G
- a **style cost function** $J_{\text{style}}(S, G)$ that measures the similarity between the styles of S and G

Then the total cost is defined as the linear combination of the two

$$J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G)$$

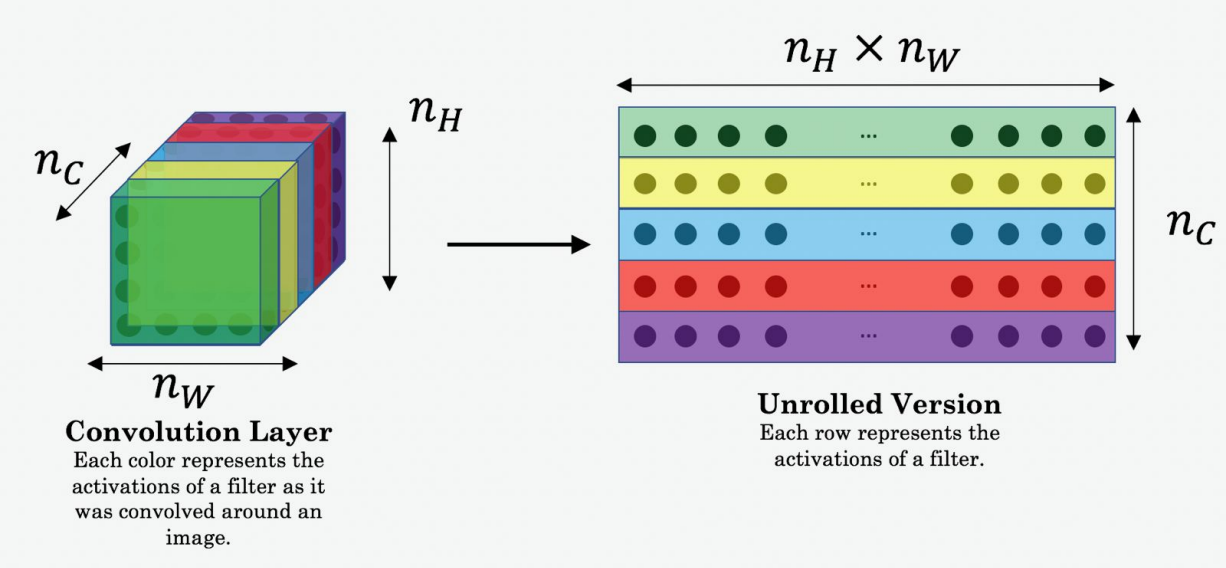
Content Cost Function $J_{\text{content}}(C, G)$

The generated image G should match the content of image C. The content cost function can be defined as:

$$J_{\text{content}}(C, G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{(C)} - a^{(G)})^2 \tag{1}$$

Here, n_H, n_W and n_C are the height, width and number of channels of the intermediate hidden layer we have chosen, and appear as a normalization term in the cost. The terms $a^{(C)}$ and $a^{(G)}$ are the 3D volumes corresponding to a hidden layer's activations of the neural network. The content cost then measures how different $a^{(C)}$ and $a^{(G)}$ are. When we minimize the content cost later, this will help make sure G has similar contents as C.

In order to compute the cost $J_{\text{content}}(C, G)$, it might also be convenient to *unroll* these 3D volumes into a 2D matrix, as shown below. Technically this unrolling step is not necessary for computing J_{content}, but it will be good practice for when we carry out a similar operation later for computing the style cost J_{style}.



Style Matrix

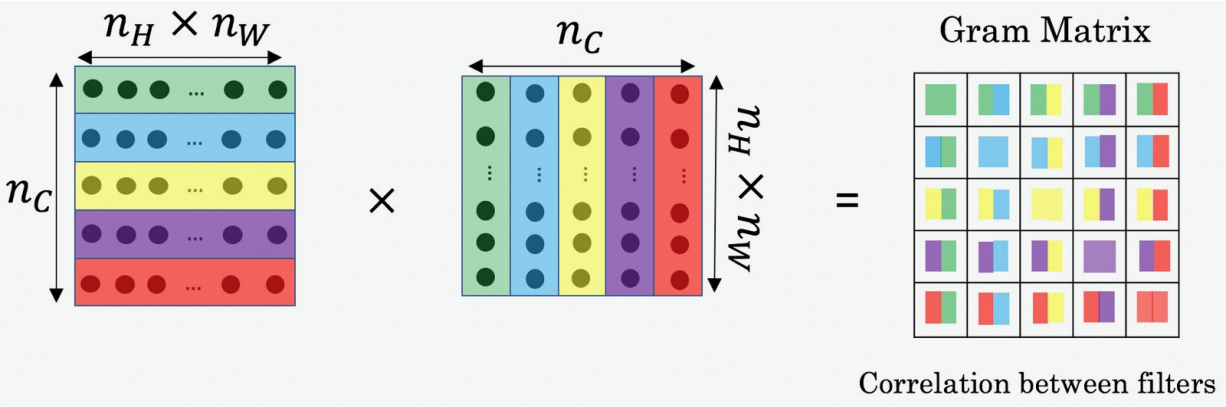
The style of an image can be represented using the Gram matrix of an intermediate layer's activations.

In linear algebra, the **Gram matrix** $\mathbf{G}$ of a set of vectors $(v_1, \ldots, v_n)$ is the matrix of dot products, whose entries are

$$\mathbf{G}_{ij} = v_i^T v_j = \text{np.dot}(v_i, v_j)$$

In NST, we compute the Style matrix by multiplying the unrolled filter matrix with its transpose

$$\mathbf{G}_{\text{gram}} = \mathbf{A}_{\text{unrolled}} \mathbf{A}_{\text{unrolled}}^T$$



In components, supopse the activation at row i, column j and channel k at a chosen hidden layer is represented by $a_{i,j,k}$, then the matrix element of $\mathbf{G}_{\text{gram}}$ is:

$$\mathbf{G}_{(\text{gram})k,k'} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a_{i,j,k}\, a_{i,j,k'}$$

The result will be a matrix of dimension $n_C \times n_C$ where n_C is the number of channels of that intermediate layer.

The matrix element $\mathbf{G}_{k,k'}$ compares how similar the activations from the filter k is to those activations from the filter k'. If a_k and $a_{k'}$ are highly correlated, their dot product should be large and thus we expect the matrix element $\mathbf{G}_{k,k'}$ to be large.

Also, the diagonal elements $\mathbf{G}_{k,k}$ measures the level of activity of the filter k. If $\mathbf{G}_{k,k}$ is large, then this means the image contains a lot of features as represented by the channel k.

By capturing the prevalence of different types of features ($\mathbf{G}_{k,k}$) as well as how much different features occur together ($\mathbf{G}_{k,k'}$), the matrix $\mathbf{G}_{\text{gram}}$ measures the style of the image.

Style Cost

After calculating the Gram matrix, the next goal is to minimize the distance between the Gram matrix of the style image S and the Gram matrix of the generated image G.

For a single hidden layer, we can compute the Gram matrices of the style image S and the generated image G, that is $\mathbf{G}_{\text{gram}}^{(S)}$ and $\mathbf{G}_{\text{gram}}^{(G)}$, using the activations $a^{[l]}$ from this particular intermediate layer in the network. The corresponding style cost is defined as:

$$J_{\text{style}}^{[l]}(S,G) = \frac{1}{4 \times {n_C}^2 \times (n_H \times n_W)^2} \sum_{k=1}^{n_C} \sum_{k'=1}^{n_C} (\mathbf{G}_{(\text{gram})k,k'}^{(S)} - \mathbf{G}_{(\text{gram})k,k'}^{(G)})^2 \tag{2}$$

Better results can be obtained if the representation of the style image S is computed from multiple different layers and combined. The style costs from several different layers can be merged to give:

$$J_{\text{style}}(S,G) = \sum_{l} \lambda^{[l]} J_{\text{style}}^{[l]}(S,G)$$

where $\lambda^{[l]}$ are the weights that reflect how much each layer contributes to the style and they are subject to the normalisation condition $\sum_{l=1}^{L} \lambda^{[l]} = 1$.

The choice of the coefficients for each layer depends on how much we want the generated image to follow the style image. Since deeper layers capture higher-level concepts, and the features in the deeper layers are less localized in the image relative to each other, so if we want to follow the style image softly, larger weights for deeper layers and smaller weights for the earlier layers can be chosen. In contrast, if we want the generated image to strongly follow the style image, we may choose smaller weights for deeper layers and larger weights for the earlier layers.

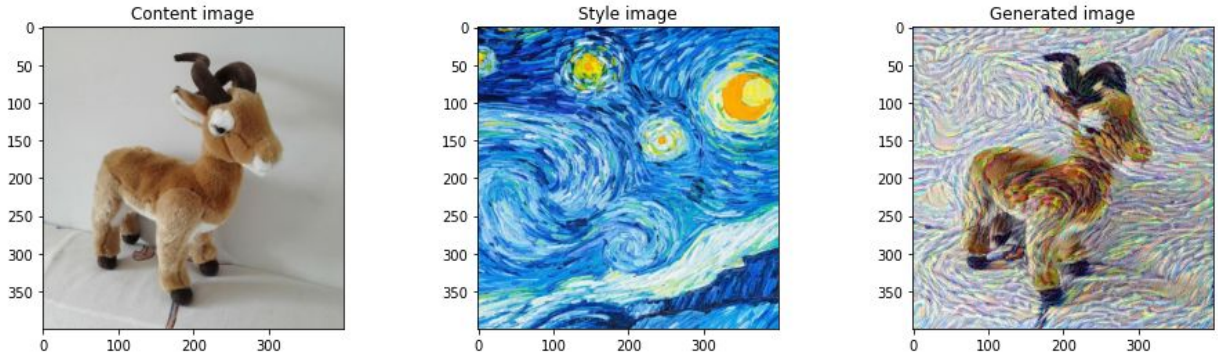
Total Cost

The content cost and the style cost functions are combined to give the total cost:

$$J(G) = \alpha J_{\text{content}}(C,G) + \beta J_{\text{style}}(S,G)$$

Here α and β are hyperparameters that control the relative weighting between content and style.

By loading the VGG19 model and choosing a suitable optimizer for the total cost, we are able to implement Neural Style Transfer and generate artistic images with lots of fun!



Course 4-1A: Recurrent Neural Networks

Sequence models allow AI systems to understand and make predictions based on **time-series data**. The order of the data points in such tasks is important for generate outputs. Examples are natural language processing, speech recognition, and music generation.

Notations and Representation of Words

Suppose we want the model to analyse a sentence and identify the names in it, so the inputs are the words and the outputs are binary that tell if each word is a name or not. For example, if the input sentence is:

$$\text{Harry Potter and Hermonie Granger invented a new spell.}$$

Then the expected output of the named-entity recognition task is:

$$1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0$$

To represent each individual word in the input sequence, we can choose a dictionary containing all the words of interest, and then use one-hot representation for each word. If we encounter a word that is not in the vocabulary, we can represent the unknown work with the `<UNK>` token.

For such chronologically ordered data, we may index the inputs and outputs as

$$\mathbf{x}^{(i)\langle t\rangle} \qquad \hat{\mathbf{y}}^{(i)\langle t\rangle}$$

- Superscript (i) denotes an object associated with the i^{th} training example.
- Superscript $\langle t\rangle$ denotes an object at the t^{th} time step.
- Superscript $[l]$ denotes an object associated with the l^{th} layer.
- Subscript j denotes the j^{th} entry of a vector.

Recurrent Neural Networks

Why Not Standard Neural Networks?

Problems with a standard neural network for sequential tasks include:

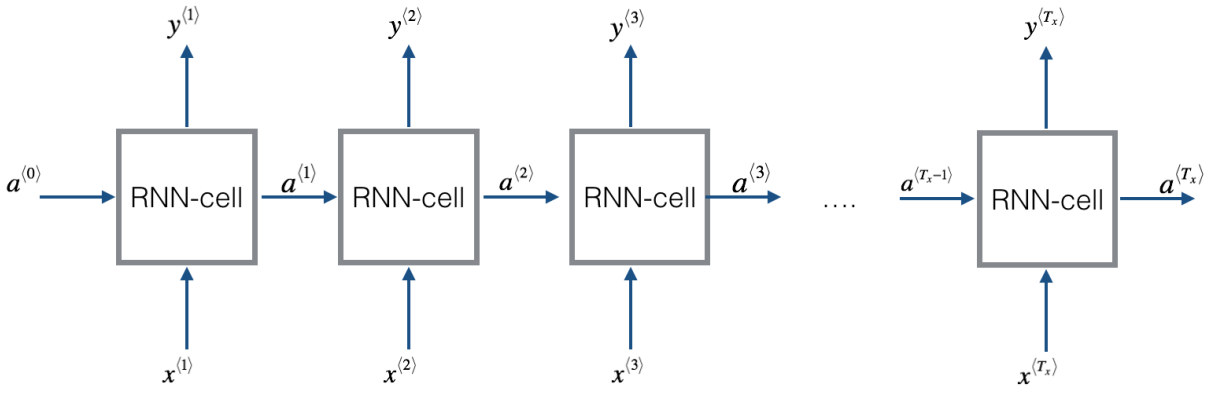
- Inputs and outputs may be different in lengths.

  This can be solved by padding with the maximum length, but it is not always a good solution.
- Features learned across different positions of the sequence are not shared.

  For example, in name entity recognition tasks, a person's name reappearing in a different position simply means the same word is also a name.
- The length of the inputs can be very big, depending on the size of the dictionary, so the network can have a huge number of parameters.

Features of Recurrent Neural Networks

Recurrent Neural Networks (RNNs) effective for natural language processing and other sequence tasks because they have "memory". They do so by maintaining a **hidden state** $\mathbf{a}^{\langle t\rangle}$ that stores information from previous steps. Instead of processing each input independently, RNNs use recurrent connections to pass information from one step to the next. RNNs process one word at a time step, update the corresponding hidden state up to that point. This allows an RNN to process sequences of arbitrary lengths.



The discussions above apply to what we call a **uni-directional** (one-way) RNN, which means that the RNN can take information from the past to process later inputs, i.e., only forward activations are used to compute the output:

$$\hat{\mathbf{y}}^{\langle t\rangle} = g(\mathbf{W}_{ya}\mathbf{a}^{\langle t\rangle} + \mathbf{b}_y)$$

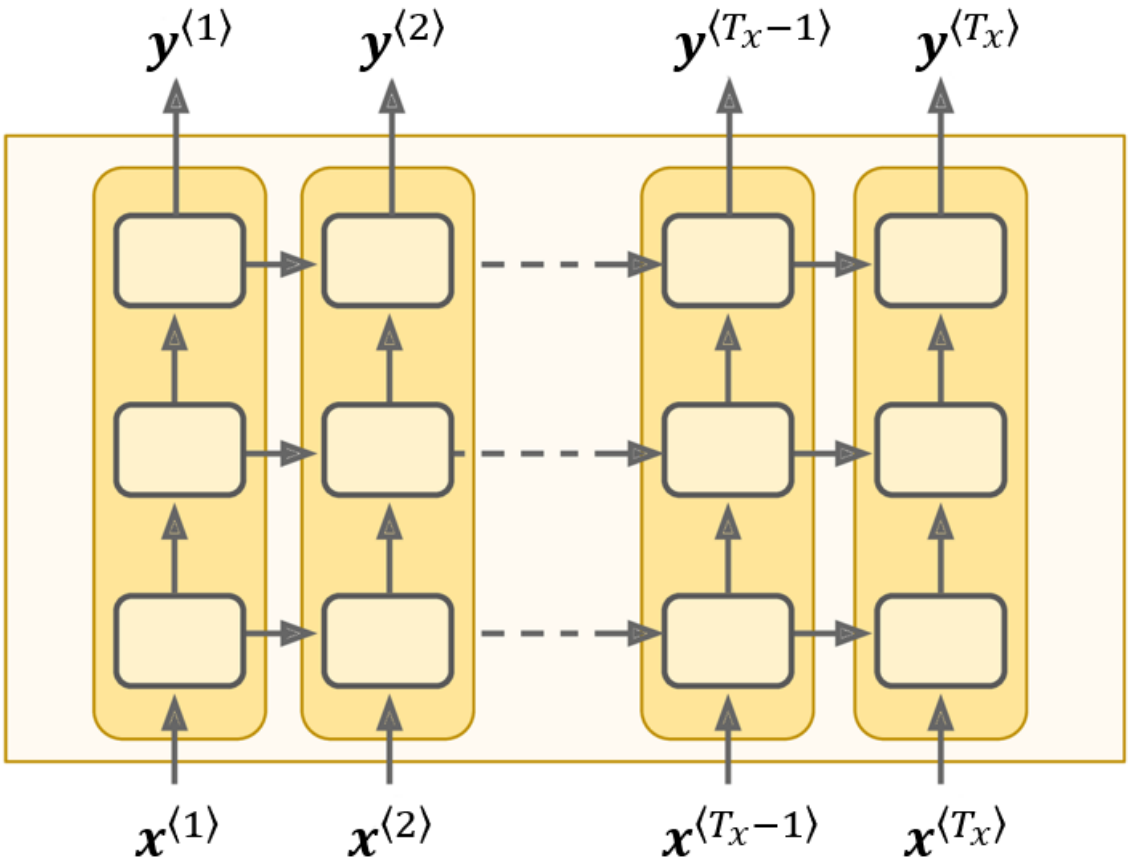
It is also possible to build **bidirectional** (two-way) RNNs to take context from both the past and the future. In this case, both the forward and backward activations are used to compute the output:

$$\hat{\mathbf{y}}^{\langle t\rangle} = g(\mathbf{W}_{ya}[\mathbf{a}^{\langle t\rangle}, \mathbf{a}'^{\langle t\rangle}] + \mathbf{b}_y)$$

Another key mechanism in RNNs is the repeated use of the **same set of weight parameters** across all time steps. At each time step, the network takes two inputs: the current data point $\mathbf{x}^{\langle t\rangle}$ (such as words) and the hidden state from the previous step $\mathbf{a}^{\langle t-1\rangle}$. These two sets of inputs are combined using the same weights and activation functions to produce a new hidden state and an output. Such parameter sharing makes RNNs more efficient than

standard neural networks for sequential tasks.

For learning very complex functions, it is useful to stack multiple layers of RNNs to build deeper versions of these models. For each time step, we can stack many hidden layers and build lateral connections between them. A deep RNN with three layers would look like this:



Types of RNNs

So far we are focussing an RNN architecture such that size of the input sequence T_x is equal to the size of the output sequence T_y. But in many problems, T_x and T_y can be different, leading to the need of different architectures.

- Many to many, like **machine translation** (would need two parts, an encoder and a decoder)
- Many to one, like **sentiment analysis** (input a sequence and output a sentiment such as positive, negative or neutral)
- One to many, like **music generation** (input one note and the network generates a series of music notes, and reuses the outputs as new inputs to generate further music notes)

Problems with RNNs

One of the major issue with RNNs is the **vanishing and exploding gradient problem**. We can have very **long dependencies**, like in the sentence "the *person/people* who ..... *is/are* ", depending on the subject being singular or plural, the network needs to output the correct prediction much later in the sequence.

To learn such long dependencies, the gradient needs to backpropagate over very large number of steps to affect the earlier layers. But this is not always possible because of the problem of vanishing and exploding gradients. Therefore, we only have local dependencies in RNNs, where each word only depend on a limited number of words preceding it.

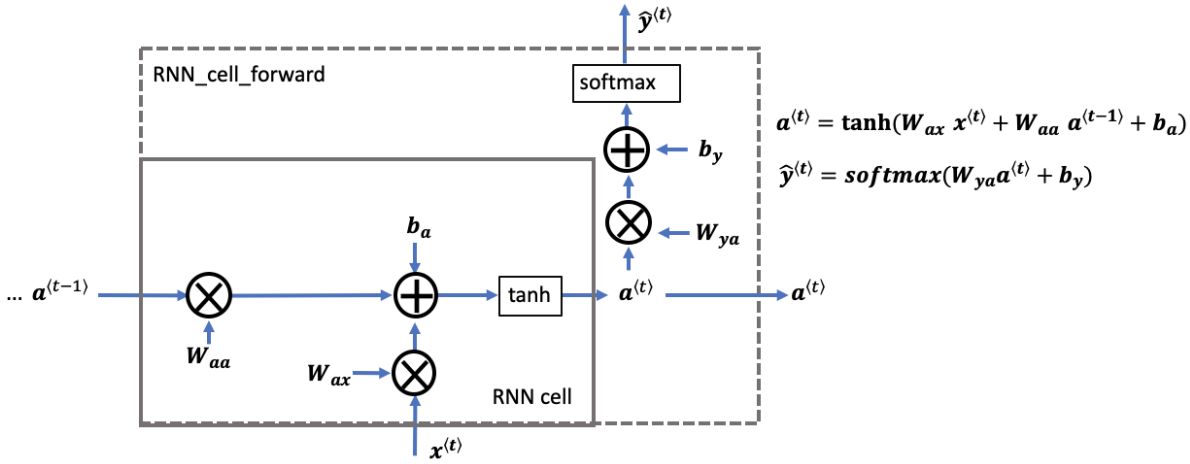
For exploding gradient we can apply **gradient clipping**. That is to clip any overly large values of gradients according to some threshold value whenever they appear to explode during forward propagation.

Vanishing gradient is a much trickier problem. It could be mitigated with better **weight initialisation** schemes, and new architectures such as **LSTM/GRU** networks that we will discuss later on.

Basic RNN Cells

Forward Propagation

We can think of the RNN as the repeated use of a single cell. The following figure describes the operations during a single time step of a basic RNN cell.

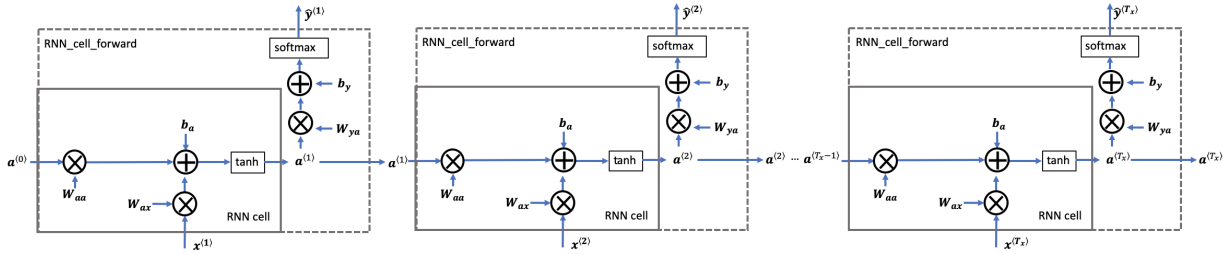


Each basic RNN cell takes as input $\mathbf{x}^{\langle t \rangle}$ (input from the current time step) and $\mathbf{a}^{\langle t-1 \rangle}$ (previous hidden state that contains information from the past time steps), and outputs a prediction $\hat{\mathbf{y}}^{\langle t \rangle}$ for the current time step together with a new hidden state $\mathbf{a}^{\langle t \rangle}$ which is passed on to the next RNN cells.

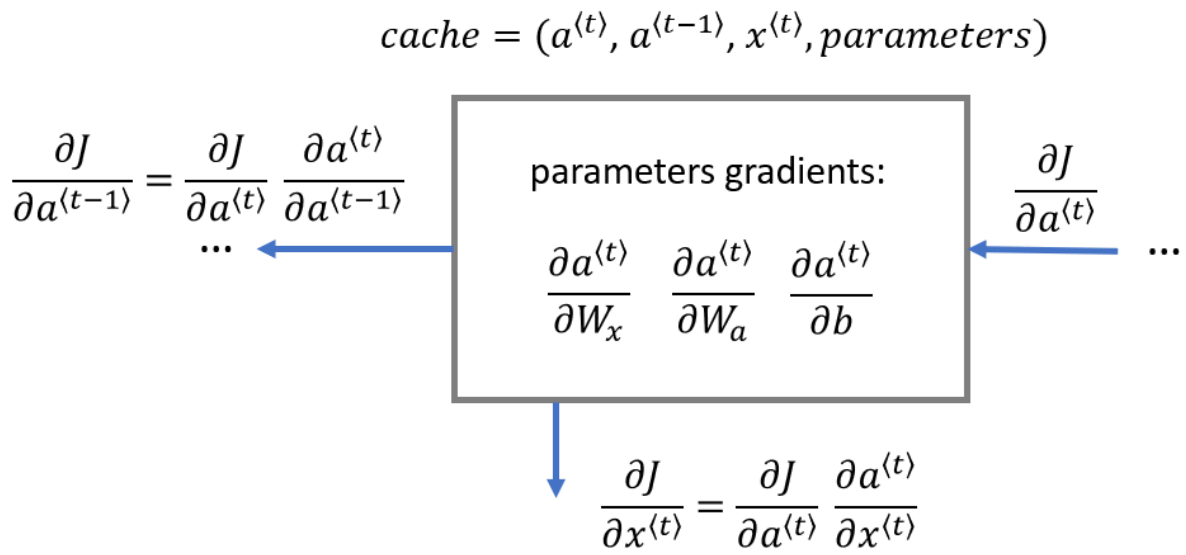
$$\mathbf{a}^{\langle t \rangle} = \tanh(\mathbf{W}_{aa}\mathbf{a}^{\langle t-1 \rangle} + \mathbf{W}_{ax}\mathbf{x}^{\langle t \rangle} + \mathbf{b}_a)$$
$$\hat{\mathbf{y}}^{\langle t \rangle} = \text{softmax}(\mathbf{W}_{ya}\mathbf{a}^{\langle t \rangle} + \mathbf{b}_y)$$

Note that the hyperbolic $\tanh$ is chosen as the activation function so that $\mathbf{a}^{\langle t \rangle}$ is always a tensor containing values within the range $(-1, 1)$.

A recurrent neural network (RNN) is a repetition of the basic RNN cells, where the weights and biases $(\mathbf{W}_{aa}, \mathbf{W}_{ax}, \mathbf{b}_a, \mathbf{W}_{ay}, \mathbf{b}_y)$ are reused at each time step.



Backward Propagation



For convenience, let's introduce the pre-activation variable:

$$\mathbf{z}^{\langle t \rangle} = \mathbf{W}_{ax}\mathbf{x}^{\langle t \rangle} + \mathbf{W}_{aa}\mathbf{a}^{\langle t-1 \rangle} + \mathbf{b}_a$$

We can then write the output of the basic cell as:

$$\mathbf{a}^{\langle t \rangle} = \tanh(\mathbf{z}^{\langle t \rangle})$$

We start deriving the partial derivatives of the cost functions with the following:

$$\begin{aligned} d\tanh = \frac{\partial J}{\partial \mathbf{z}^{\langle t \rangle}} &= \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}} \frac{\partial \mathbf{a}^{\langle t \rangle}}{\partial \mathbf{z}^{\langle t \rangle}} \\ &= d\mathbf{a}^{\langle t \rangle} * \left(1 - \tanh^2(\mathbf{z}^{\langle t \rangle})\right) \\ \Rightarrow d\tanh &= d\mathbf{a}^{\langle t \rangle} * \left(1 - (\mathbf{a}^{\langle t \rangle})^2\right) \end{aligned}$$

Here I am using the notation from Andrew's course, but I personally find it more pleasing to change $d\tanh$ into $d\mathbf{z}^{\langle t \rangle}$. Anyway, we may then proceed to find the partial derivatives with respect to the weights, biases and outputs from the previous layer:

$$\begin{aligned} d\mathbf{W}_{ax} &= \frac{\partial J}{\partial \mathbf{z}^{\langle t \rangle}} \frac{\partial \mathbf{z}^{\langle t \rangle}}{\partial \mathbf{W}_{ax}} = d\tanh \cdot \mathbf{x}^{\langle t \rangle T} \\ d\mathbf{W}_{aa} &= \frac{\partial J}{\partial \mathbf{z}^{\langle t \rangle}} \frac{\partial \mathbf{z}^{\langle t \rangle}}{\partial \mathbf{W}_{aa}} = d\tanh \cdot \mathbf{a}^{\langle t-1 \rangle T} \\ d\mathbf{b}_a &= \frac{\partial J}{\partial \mathbf{u}^{\langle t \rangle}} \frac{\partial \mathbf{z}^{\langle t \rangle}}{\partial \mathbf{b}_a} = \sum_{\text{batch}} d\tanh \\ d\mathbf{x}^{\langle t \rangle} &= \frac{\partial J}{\partial \mathbf{z}^{\langle t \rangle}} \frac{\partial \mathbf{z}^{\langle t \rangle}}{\partial \mathbf{x}^{\langle t \rangle}} = \mathbf{W}_{ax}{}^T \cdot d\tanh \\ d\mathbf{a}^{\langle t-1 \rangle} &= \frac{\partial J}{\partial \mathbf{z}^{\langle t \rangle}} \frac{\partial \mathbf{z}^{\langle t \rangle}}{\partial \mathbf{a}^{\langle t-1 \rangle}} = \mathbf{W}_{aa}{}^T \cdot d\tanh \end{aligned}$$

We may check that these parameter gradients make sense by checking the dimensions of the vectors and tensors that appear on both sides:

$$\begin{aligned} d\mathbf{W}_{ax} &: \quad (n_a \times n_x) = (n_a \times 1) \cdot (1 \times n_x) \\ d\mathbf{W}_{aa} &: \quad (n_a \times n_a) = (n_a \times 1) \cdot (1 \times n_a) \\ d\mathbf{b}_a &: \quad (n_a \times 1) = (n_a \times 1) \\ d\mathbf{x}^{\langle t \rangle} &: \quad (n_x \times 1) = (n_x \times n_a) \cdot (n_a \times 1) \\ d\mathbf{a}^{\langle t-1 \rangle} &: \quad (n_a \times 1) = (n_a \times n_a) \cdot (n_a \times 1) \end{aligned}$$

Course 4-1B: LSTM & GRU Networks

Long Short-Term Memory (LSTM) Networks: Forward Propagation

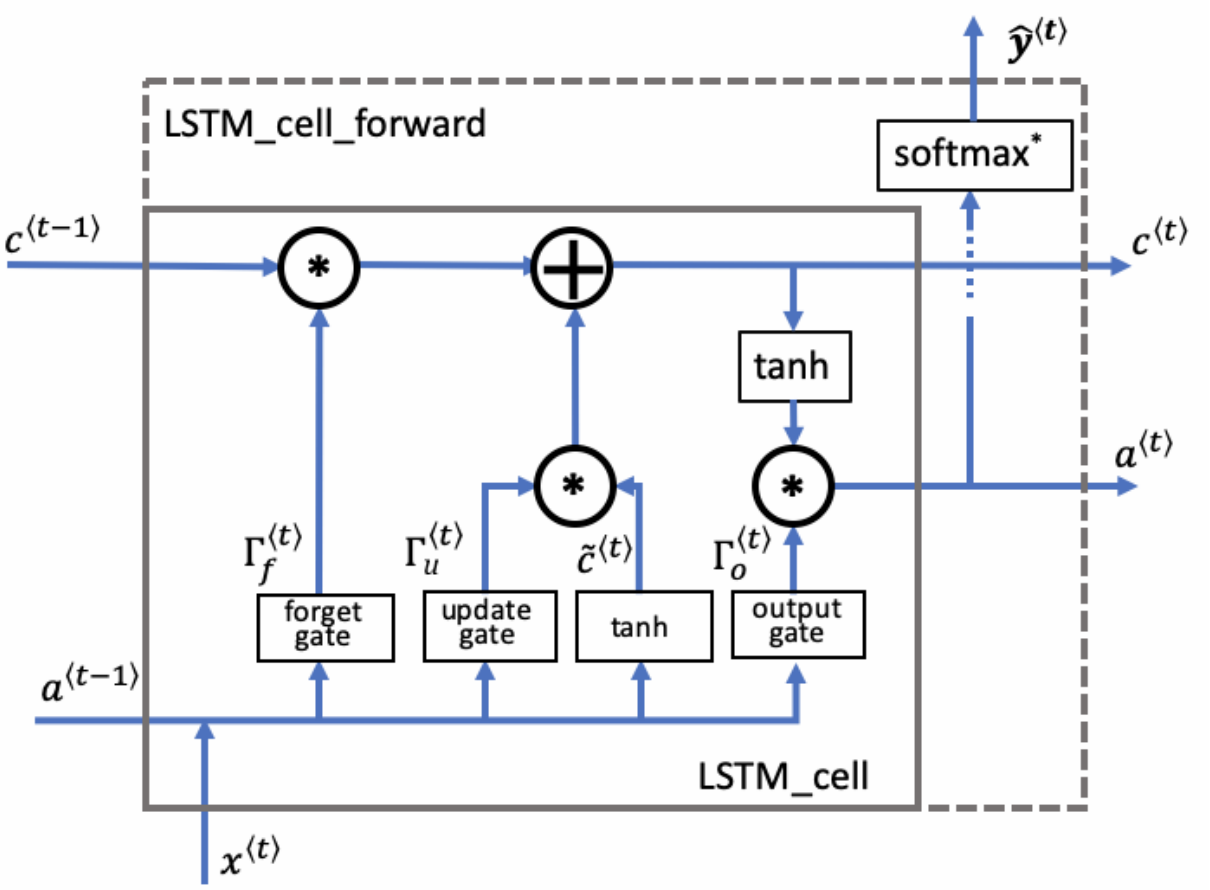
Long Short-Term Memory (LSTM) networks are a specialized type of RNN designed to learn long-range dependencies in sequential data, solving the vanishing gradient problem encountered by traditional RNNs. They use a **gating mechanism** to selectively remember or forget information over long time steps, making them ideal for learning time series data.

An LSTM is similar to an RNN in that they both use hidden states to pass along information. In addition to hidden states $\mathbf{a}^{\langle t \rangle}$, an LSTM network tracks and updates a **cell state**, or **memory variable**, $\mathbf{c}^{\langle t \rangle}$ at every time step, which can be different from $\mathbf{a}^{\langle t \rangle}$. Each $\mathbf{c}^{\langle t \rangle}$ provides a bit of extra memory to remember and is passed on along the network.

The key ingredients of a LSTM cell include

- a **cell state** $\mathbf{c}^{\langle t \rangle}$, which is like a long-term memory
- a **hidden state** $\mathbf{a}^{\langle t \rangle}$, which is like a short-term memory
- three gates (**forget gate**, **update gate** and **output gate**), that regulate what information to be stored or discarded depending on the relevancy of the inputs

The following figure illustrates the operations for a single time step of an LSTM cell.



Forget gate $\mathbf{\Gamma}_f$

$$\mathbf{\Gamma}_f^{\langle t \rangle} = \sigma(\mathbf{W}_f[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_f)$$

The forget gate is a tensor containing values between 0 and 1. To calculate this tensor, we first take a linear combination of $\mathbf{a}^{\langle t-1 \rangle}$ (hidden state of the previous time step) and $\mathbf{x}^{\langle t \rangle}$ (input of the current time step), and then apply a sigmoid function to make each of the tensor's values range from 0 to 1 as desired.

The forget gate helps us keep track of grammatical structures such as whether the subject in a piece of text is singular or plural. If the subject changes its state, say it changes from a singular word to a plural word, then the memory of the previous state becomes outdated, the unit in the forget gate will get a value close to 0 and the LSTM will forget the outdated state $\mathbf{c}^{\langle t-1 \rangle}$ that has been stored in the previous cell. If the unit the forget state gets a value close to 1, the LSTM will mostly remember the store value in the corresponding unit $\mathbf{c}^{\langle t-1 \rangle}$.

Candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$

$$\tilde{\mathbf{c}}^{\langle t \rangle} = \tanh\left(\mathbf{W}_c [\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_c \right)$$

The candidate value is a tensor containing values between -1 and 1, which is guaranteed by the range of the $\tanh$ function.

The candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$ contains information from the current time step that may be stored in the current cell state $\mathbf{c}^{\langle t \rangle}$. Some part of the candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$ gets passed on to $\mathbf{c}^{\langle t \rangle}$, depending on the update gate $\mathbf{\Gamma}_u^{\langle t \rangle}$.

Update gate $\mathbf{\Gamma}_u$

$$\mathbf{\Gamma}_u^{\langle t \rangle} = \sigma(\mathbf{W}_u [\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_u)$$

Similar to the forget gate, the update gate produces values between 0 and 1.

The update date decides which part of the candidate tensor $\tilde{\mathbf{c}}^{\langle t \rangle}$ gets passed onto the new cell state $\mathbf{c}^{\langle t \rangle}$. When a unit in the update gate is close to 1, it allows the value of the candidate $\tilde{\mathbf{c}}^{\langle t \rangle}$ to be passed onto the cell state $\mathbf{c}^{\langle t \rangle}$. When a unit in the update gate is close to 0, it prevents the corresponding value in the candidate from being passed on.

Cell state $\mathbf{c}^{\langle t \rangle}$

$$\mathbf{c}^{\langle t \rangle} = \mathbf{\Gamma}_f^{\langle t \rangle} * \mathbf{c}^{\langle t-1 \rangle} + \mathbf{\Gamma}_u^{\langle t \rangle} * \tilde{\mathbf{c}}^{\langle t \rangle}$$

The new cell state $\mathbf{c}^{\langle t \rangle}$ is a linear combination of the previous cell state $\mathbf{c}^{\langle t-1 \rangle}$ and the candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$. The weights of this linear combination are exactly the forget gate $\mathbf{\Gamma}_f^{\langle t \rangle}$ and the update gate $\mathbf{\Gamma}_u^{\langle t \rangle}$ that we have introduced above.

We can think of the cell state $\mathbf{c}^{\langle t \rangle}$ as the long-term memory that gets passed on to future time steps.

Output gate $\mathbf{\Gamma}_o$

$$\mathbf{\Gamma}_o^{\langle t \rangle} = \sigma(\mathbf{W}_o [\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_o)$$
$$\mathbf{a}^{\langle t \rangle} = \mathbf{\Gamma}_o^{\langle t \rangle} * \tanh(\mathbf{c}^{\langle t \rangle})$$

Like the other gates, values of the output gate are always between 0 to 1.

The output gate decides what fraction of the cell state gets exposed as the hidden state $\mathbf{a}^{\langle t \rangle}$ of the current time step.

Prediction $\mathbf{y}_{\text{pred}}^{\langle t \rangle}$

$$\mathbf{y}_{\text{pred}}^{\langle t \rangle} = \text{softmax}(\mathbf{W}_y \mathbf{a}^{\langle t \rangle} + \mathbf{b}_y)$$

The prediction in this our case is a classification, so a softmax function is used.

Summary

In a nutshell, the algebraic operations for the forward pass are:

- Forget gate: $\mathbf{\Gamma}_f^{\langle t \rangle} = \sigma(\mathbf{W}_f [\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_f)$
- Candidate value: $\tilde{\mathbf{c}}^{\langle t \rangle} = \tanh\left(\mathbf{W}_c [\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_c \right)$
- Update gate: $\mathbf{\Gamma}_u^{\langle t \rangle} = \sigma(\mathbf{W}_u [\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_u)$
- Cell state: $\mathbf{c}^{\langle t \rangle} = \mathbf{\Gamma}_f^{\langle t \rangle} * \mathbf{c}^{\langle t-1 \rangle} + \mathbf{\Gamma}_u^{\langle t \rangle} * \tilde{\mathbf{c}}^{\langle t \rangle}$
- Output gate: $\mathbf{\Gamma}_o^{\langle t \rangle} = \sigma(\mathbf{W}_o [\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_o)$
- Hidden state: $\mathbf{a}^{\langle t \rangle} = \mathbf{\Gamma}_o^{\langle t \rangle} * \tanh(\mathbf{c}^{\langle t \rangle})$
- Prediction: $\mathbf{y}_{\text{pred}}^{\langle t \rangle} = \text{softmax}(\mathbf{W}_y \mathbf{a}^{\langle t \rangle} + \mathbf{b}_y)$

Long Short-Term Memory (LSTM) Networks: Backward Propagation

In this section, we give a step-by-step derivation for the derivatives for backward propagation at the level of each individual gate.

Gate Derivatives

For convenience, let's introduce pre-activation variables

$$\begin{aligned}
\mathbf{z}_o^{\langle t \rangle} &= \mathbf{W}_o[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_o \\
\mathbf{z}_u^{\langle t \rangle} &= \mathbf{W}_u[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_u \\
\mathbf{z}_f^{\langle t \rangle} &= \mathbf{W}_f[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_f \\
\tilde{\mathbf{z}}^{\langle t \rangle} &= \mathbf{W}_c[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_c
\end{aligned}$$

So the gate values and the candidate values can be expressed nicely as

- Output gate: $\mathbf{\Gamma}_o^{\langle t \rangle} = \sigma(\mathbf{z}_o^{\langle t \rangle})$
- Candidate value: $\tilde{\mathbf{c}}^{\langle t \rangle} = \tanh\left(\tilde{\mathbf{z}}^{\langle t \rangle}\right)$
- Update gate: $\mathbf{\Gamma}_u^{\langle t \rangle} = \sigma(\mathbf{z}_u^{\langle t \rangle})$
- Forget gate: $\mathbf{\Gamma}_f^{\langle t \rangle} = \sigma(\mathbf{z}_f^{\langle t \rangle})$

Derivatives for the output gate is easy to find. Recall that for sigmoid function $\sigma(z) = \dfrac{1}{1+e^{-z}}$, we have $\sigma'(z) = \sigma(z)(1-\sigma(z))$. So we have:

$$\begin{aligned}
d\mathbf{z}_o^{\langle t \rangle} &= \frac{\partial J}{\partial \mathbf{z}_o^{\langle t \rangle}} = \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}} \frac{\partial \mathbf{a}^{\langle t \rangle}}{\partial \mathbf{\Gamma}_o^{\langle t \rangle}} \frac{\partial \mathbf{\Gamma}_o^{\langle t \rangle}}{\partial \mathbf{z}_o^{\langle t \rangle}} \\
\Rightarrow d\mathbf{z}_o^{\langle t \rangle} &= d\mathbf{a}^{\langle t \rangle} * \tanh(\mathbf{c}^{\langle t \rangle}) * \mathbf{\Gamma}_o^{\langle t \rangle} * \left(1 - \mathbf{\Gamma}_o^{\langle t \rangle}\right)
\end{aligned}$$

For the other gate derivatives, note that both the forget gate and the update gate participate in the computation of the cell state: $\mathbf{c}^{\langle t \rangle} = \mathbf{\Gamma}_f^{\langle t \rangle} * \mathbf{c}^{\langle t-1 \rangle} + \mathbf{\Gamma}_u^{\langle t \rangle} * \tilde{\mathbf{c}}^{\langle t \rangle}$. This means that when we compute these gate derivatives, there are contributions from two sources: the loss both at the current time step and also from the future time steps. Due to such recurrence structure in $\mathbf{c}^{\langle t \rangle}$, the full gradient of the cell state is:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}} &= \frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}}\bigg|_{\text{local}} + \frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}}\bigg|_{\text{future}} \\
&= \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}} \frac{\partial \mathbf{a}^{\langle t \rangle}}{\partial \mathbf{c}^{\langle t \rangle}} + \frac{\partial J}{\partial \mathbf{c}^{\langle t+1 \rangle}} \frac{\partial \mathbf{c}^{\langle t+1 \rangle}}{\partial \mathbf{c}^{\langle t \rangle}}
\end{aligned}$$

With $\mathbf{a}^{\langle t \rangle} = \mathbf{\Gamma}_o^{\langle t \rangle} * \tanh(\mathbf{c}^{\langle t \rangle})$ and $\mathbf{c}^{\langle t+1 \rangle} = \mathbf{\Gamma}_f^{\langle t+1 \rangle} * \mathbf{c}^{\langle t \rangle} + \mathbf{\Gamma}_u^{\langle t+1 \rangle} * \tilde{\mathbf{c}}^{\langle t+1 \rangle}$, we further obtain

$$d\mathbf{c}^{\langle t \rangle} = d\mathbf{a}^{\langle t \rangle} * \mathbf{\Gamma}_o^{\langle t \rangle} * \left(1 - \tanh^2(\mathbf{c}^{\langle t \rangle})\right) + d\mathbf{c}^{\langle t+1 \rangle} * \mathbf{\Gamma}_f^{\langle t+1 \rangle}$$

In implementation, we can introduce the passed-back variable $d\mathbf{c}_{\text{prev}}^{\langle t \rangle} = d\mathbf{c}^{\langle t+1 \rangle} * \mathbf{\Gamma}_f^{\langle t+1 \rangle}$ at time step $t+1$, so at time step t we receive a single tensor $d\mathbf{c}_{\text{prev}}^{\langle t \rangle}$ with the $\mathbf{\Gamma}_f^{\langle t+1 \rangle}$ factor already pre-multiplied, and we use that to compute the full gradient $d\mathbf{c}^{\langle t \rangle}$ at time step t, i.e.,

$$\frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}} = d\mathbf{a}^{\langle t \rangle} * \mathbf{\Gamma}_o^{\langle t \rangle} * \left(1 - \tanh^2(\mathbf{c}^{\langle t \rangle})\right) + d\mathbf{c}_{\text{prev}}^{\langle t \rangle}$$

Now, for the candidate value:

$$\begin{aligned}
d\mathbf{z}_c^{\langle t \rangle} &= \frac{\partial J}{\partial \mathbf{z}_c^{\langle t \rangle}} = \frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}} \frac{\partial \mathbf{c}^{\langle t \rangle}}{\partial \tilde{\mathbf{c}}^{\langle t \rangle}} \frac{\partial \tilde{\mathbf{c}}^{\langle t \rangle}}{\partial \mathbf{z}_c^{\langle t \rangle}} \\
&= \frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}} * \mathbf{\Gamma}_u^{\langle t \rangle} * \left(1 - \tanh^2(\mathbf{z}_c^{\langle t \rangle})\right) \\
\Rightarrow d\mathbf{z}_c^{\langle t \rangle} &= d\mathbf{c}^{\langle t \rangle} * \mathbf{\Gamma}_u^{\langle t \rangle} * \left(1 - (\tilde{\mathbf{c}}^{\langle t \rangle})^2\right)
\end{aligned}$$

Derivatives for the update gate follows similarly:

$$
\begin{aligned}
d\mathbf{z}_u^{\langle t \rangle} &= \frac{\partial J}{\partial \mathbf{z}_u^{\langle t \rangle}} = \frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}} \frac{\partial \mathbf{c}^{\langle t \rangle}}{\partial \mathbf{\Gamma}_u^{\langle t \rangle}} \frac{\partial \mathbf{\Gamma}_u^{\langle t \rangle}}{\partial \mathbf{z}_u^{\langle t \rangle}} \\
&= \frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}} * \tilde{\mathbf{c}}^{\langle t \rangle} * \left[\sigma(\mathbf{z}_u^{\langle t \rangle}) * (1 - \sigma(\mathbf{z}_u^{\langle t \rangle})) \right] \\
\Rightarrow d\mathbf{z}_u^{\langle t \rangle} &= d\mathbf{c}^{\langle t \rangle} * \tilde{\mathbf{c}}^{\langle t \rangle} * \mathbf{\Gamma}_u^{\langle t \rangle} * \left(1 - \mathbf{\Gamma}_u^{\langle t \rangle} \right)
\end{aligned}
$$

Also, derivative for the forget gate is almost identical:

$$
\begin{aligned}
d\mathbf{z}_f^{\langle t \rangle} &= \frac{\partial J}{\partial \mathbf{z}_f^{\langle t \rangle}} = \frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}} \frac{\partial \mathbf{c}^{\langle t \rangle}}{\partial \mathbf{\Gamma}_f^{\langle t \rangle}} \frac{\partial \mathbf{\Gamma}_f^{\langle t \rangle}}{\partial \mathbf{z}_f^{\langle t \rangle}} \\
\Rightarrow d\mathbf{z}_f^{\langle t \rangle} &= d\mathbf{c}^{\langle t \rangle} * \mathbf{c}^{\langle t-1 \rangle} * \mathbf{\Gamma}_f^{\langle t \rangle} * \left(1 - \mathbf{\Gamma}_f^{\langle t \rangle} \right)
\end{aligned}
$$

Parameter Derivatives

Derivations for the parameter derivatives are straightforward:

$$
\begin{aligned}
d\mathbf{W}_* &= d\mathbf{z}_*^{\langle t \rangle} \begin{bmatrix} \mathbf{a}^{\langle t-1 \rangle} \\ \mathbf{x}^{\langle t \rangle} \end{bmatrix}^T \\
d\mathbf{b}_* &= \sum_{\text{batch}} d\mathbf{z}_*^{\langle t \rangle}
\end{aligned}
$$

where the subscript ""$*$"" stands for f, u, o, c in turn.

Derivatives with Respect to Previous States

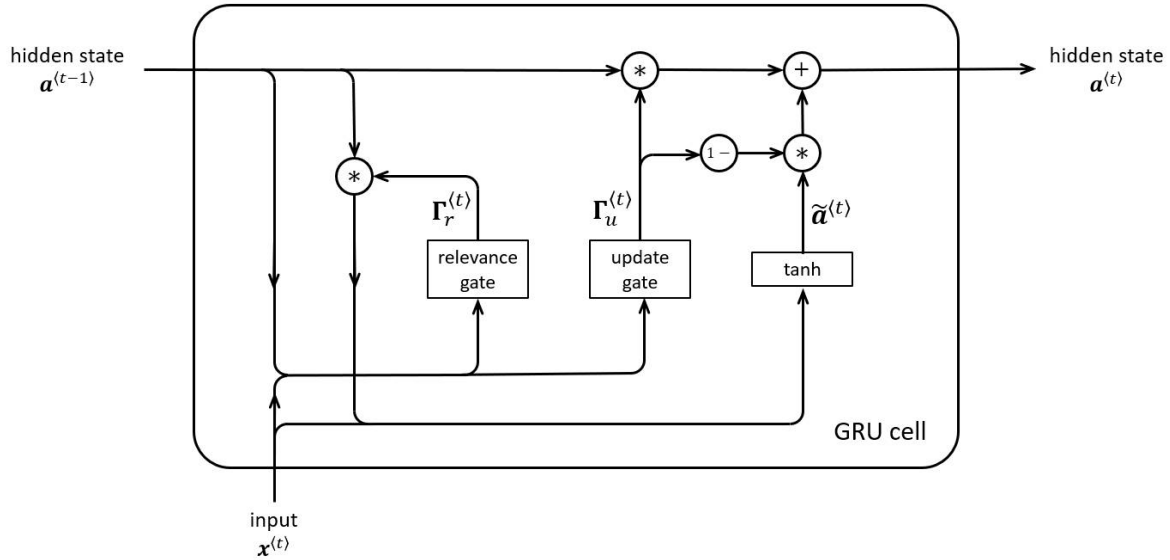
The derivatives with respect to the previous hidden state, the previous memory state and the input are:

$$
\begin{aligned}
d\mathbf{a}^{\langle t-1 \rangle} &= \mathbf{W}_f^T d\mathbf{z}_f^{\langle t \rangle} + \mathbf{W}_u^T d\mathbf{z}_u^{\langle t \rangle} + \mathbf{W}_c^T d\mathbf{z}_c^{\langle t \rangle} + \mathbf{W}_o^T d\mathbf{z}_o^{\langle t \rangle} \\
d\mathbf{c}_{\text{prev}}^{\langle t-1 \rangle} &= \frac{\partial J}{\partial \mathbf{c}^{\langle t \rangle}} \frac{\partial \mathbf{c}^{\langle t \rangle}}{\partial \mathbf{c}^{\langle t-1 \rangle}} = d\mathbf{c}^{\langle t \rangle} * \mathbf{\Gamma}_f^{\langle t \rangle}
\end{aligned}
$$

Note that the $d\mathbf{c}_{\text{prev}}^{\langle t-1 \rangle}$ being computed here is not the full derivative but the passed-back variable that only takes into account the future contributions the loss function. As we have discussed earlier, this future term needs to be combined with the local terms to assemble the full gradient $d\mathbf{c}^{\langle t-1 \rangle}$.

Gated Recurrent Unit (GRU) Networks: Forward Propagation

Gated Recurrent Units (GRUs) are another type of RNN architecture designed to solve vanishing gradient problems. The architecture of GRU networks are similar to LSTMs but with fewer gates and states. As simplified alternative to LSTMs, GRUs are usually faster and require less memory but are often comparable in performance.



The key ingredients of a GRU cell include

- a single **update gate** $\mathbf{\Gamma}_u$ that combines the update and forget gates in LSTM

At every time step, we compute a candidate value $\tilde{\mathbf{a}}^{\langle t \rangle}$ for the new hidden state, but it is up to the gate $\mathbf{\Gamma}_u$ to decide whether or not to update the hidden state with the candidate.

- a **relevance gate** $\mathbf{\Gamma}_r$ which tells how relative is the state of the memory cell to compute the candidate value for the memory cell

The equations for these operations are:

- Update gate: $\mathbf{\Gamma}_u^{\langle t \rangle} = \sigma(\mathbf{W}_u[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_u)$
- Relevance gate: $\mathbf{\Gamma}_r^{\langle t \rangle} = \sigma(\mathbf{W}_r[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_r)$
- Candidate hidden state: $\tilde{\mathbf{a}}^{\langle t \rangle} = \tanh\left(\mathbf{W}_a[\mathbf{\Gamma}_r^{\langle t \rangle} * \mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_a\right)$
- Hidden state: $\mathbf{a}^{\langle t \rangle} = \mathbf{\Gamma}_u^{\langle t \rangle} * \mathbf{a}^{\langle t-1 \rangle} + \left(1 - \mathbf{\Gamma}_u^{\langle t \rangle}\right) * \tilde{\mathbf{a}}^{\langle t \rangle}$

Note that compared to LSTMs, GRUs have no cell states $\mathbf{c}^{\langle t \rangle}$ but hidden states $\mathbf{a}^{\langle t \rangle}$ only.

Gated Recurrent Unit (GRU) Networks: Backward Propagation

The equations for backward pass for GRUs are derived analogously to those done for the LSTM.

Again, for convenience, we introduce the following pre-activation variables

$$\begin{aligned}
\mathbf{z}_u^{\langle t \rangle} &= \mathbf{W}_u[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_u \\
\mathbf{z}_r^{\langle t \rangle} &= \mathbf{W}_r[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_r \\
\mathbf{z}_a^{\langle t \rangle} &= \mathbf{W}_a[\mathbf{\Gamma}_r^{\langle t \rangle} * \mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_a
\end{aligned}$$

Recall that $\mathbf{a}^{\langle t \rangle} = \mathbf{\Gamma}_u^{\langle t \rangle} * \mathbf{a}^{\langle t-1 \rangle} + \left(1 - \mathbf{\Gamma}_u^{\langle t \rangle}\right) * \tilde{\mathbf{a}}^{\langle t \rangle}$, this suggests that the hidden state $\mathbf{a}^{\langle t \rangle}$ feeds into the loss at the current step and also into the next cell through both $\mathbf{z}_u^{\langle t+1 \rangle}$, $\mathbf{z}_r^{\langle t+1 \rangle}$, and $\mathbf{z}_a^{\langle t+1 \rangle}$. We can write out the full gradient as the sum from two sources:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}} &= \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}}\bigg|_{\text{local}} + \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}}\bigg|_{\text{future}} \\
&= \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}}\bigg|_{\text{local}} + \frac{\partial J}{\partial \mathbf{a}^{\langle t+1 \rangle}} \cdot \frac{\partial \mathbf{a}^{\langle t+1 \rangle}}{\partial \mathbf{a}^{\langle t \rangle}} \\
&= \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}}\bigg|_{\text{local}} + d\mathbf{a}^{\langle t+1 \rangle} * \mathbf{\Gamma}_u^{\langle t+1 \rangle}
\end{aligned}$$

In practice the $d\mathbf{a}^{\langle t \rangle}$ tensor passed back by the next cell is already the full derivative. But the discussion above does matter when we compute what to be passed back to earlier cells. We will cover the computation for $d\mathbf{a}^{\langle t-1 \rangle}$ at the end of this section.

The derivative of update gate reads:

$$\begin{aligned}
d\mathbf{z}_u^{\langle t \rangle} &= \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}} \frac{\partial \mathbf{a}^{\langle t \rangle}}{\partial \mathbf{\Gamma}_u^{\langle t \rangle}} \frac{\partial \mathbf{\Gamma}_u^{\langle t \rangle}}{\partial \mathbf{z}_u^{\langle t \rangle}} \\
\Rightarrow d\mathbf{z}_u^{\langle t \rangle} &= d\mathbf{a}^{\langle t \rangle} * \left(\mathbf{a}^{\langle t-1 \rangle} - \tilde{\mathbf{a}}^{\langle t \rangle}\right) * \mathbf{\Gamma}_u^{\langle t \rangle} * \left(1 - \mathbf{\Gamma}_u^{\langle t \rangle}\right)
\end{aligned}$$

For the derivative of candidate value, we can find:

$$\begin{aligned}
d\mathbf{z}_a^{\langle t \rangle} &= \frac{\partial J}{\partial \mathbf{a}^{\langle t \rangle}} \frac{\partial \mathbf{a}^{\langle t \rangle}}{\partial \tilde{\mathbf{a}}^{\langle t \rangle}} \frac{\partial \tilde{\mathbf{a}}^{\langle t \rangle}}{\partial \mathbf{z}_a^{\langle t \rangle}} \\
\Rightarrow d\mathbf{z}_a^{\langle t \rangle} &= d\mathbf{a}^{\langle t \rangle} * \left(1 - \mathbf{\Gamma}_u^{\langle t \rangle}\right) * \left(1 - \left(\tilde{\mathbf{a}}^{\langle t \rangle}\right)^2\right)
\end{aligned}$$

Lastly, for the relevance gate, since it injects itself into the candidate computation before the weight multiplications, so its gradient path simply passes through $d\mathbf{z}_a^{\langle t \rangle}$ but not through $d\mathbf{a}^{\langle t \rangle}$, which makes it the gate derivative without a direct counterpart in the LSTM backward pass. We have:

$$\begin{aligned}
d\mathbf{z}_r^{\langle t \rangle} &= \frac{\partial J}{\partial \mathbf{z}_a^{\langle t \rangle}} \frac{\partial \mathbf{z}_a^{\langle t \rangle}}{\partial \mathbf{\Gamma}_r^{\langle t \rangle}} \frac{\partial \mathbf{\Gamma}_r^{\langle t \rangle}}{\partial \mathbf{z}_r^{\langle t \rangle}} \\
\Rightarrow d\mathbf{z}_r^{\langle t \rangle} &= d\mathbf{z}_a^{\langle t \rangle} * \mathbf{W}_{a,a} \mathbf{\Gamma}_r^{\langle t \rangle} * \mathbf{a}^{\langle t-1 \rangle} * \mathbf{\Gamma}_r^{\langle t \rangle} * \left(1 - \mathbf{\Gamma}_r^{\langle t \rangle}\right)
\end{aligned}$$

Identical in structure to the LSTM case, we can find parameter derivatives

$$d\mathbf{W}_* = d\mathbf{z}_*^{\langle t \rangle} \begin{bmatrix} \mathbf{a}^{\langle t-1 \rangle} \\ \mathbf{x}^{\langle t \rangle} \end{bmatrix}^T$$
$$d\mathbf{b}_* = \sum_{\text{batch}} d\mathbf{z}_*^{\langle t \rangle}$$

where ""$*$"" stands for u, r, a in turn. Note that for $d\mathbf{W}_a$, the stacked input vector should use $\mathbf{\Gamma}_r^{\langle t \rangle} * \mathbf{a}^{\langle t-1 \rangle}$ in place of $\mathbf{a}^{\langle t-1 \rangle}$, reflecting the forward pass definition of $\mathbf{z}_a^{\langle t \rangle}$.

The derivatives with respect to the previous hidden state $\mathbf{a}^{\langle t-1 \rangle}$ receives contributions from three local pre-activation gradients, plus a direct path through the update gate term. This reads:

$$d\mathbf{a}^{\langle t-1 \rangle} = \mathbf{W}_{u,a}^T d\mathbf{z}_u^{\langle t \rangle} + \mathbf{W}_{r,a}^T d\mathbf{z}_r^{\langle t \rangle} + \mathbf{W}_{a,a}^T d\mathbf{z}_a^{\langle t \rangle} * \mathbf{\Gamma}_r^{\langle t \rangle} + d\mathbf{a}^{\langle t \rangle} * \mathbf{\Gamma}_u^{\langle t \rangle}$$

The last term $d\mathbf{a}^{\langle t \rangle} * \mathbf{\Gamma}_u^{\langle t \rangle}$ can be understood as the GRU analogue of the LSTM cell-state highway $d\mathbf{c}_{\text{prev}}^{\langle t-1 \rangle} = d\mathbf{c}^{\langle t \rangle} * \mathbf{\Gamma}_f^{\langle t \rangle}$. When $\mathbf{\Gamma}_u^{\langle t \rangle} \approx 1$, the hidden state is carried forward largely unchanged and this term dominates, propagating gradients back cleanly over long distances. This is the mechanism by which GRUs mitigate the vanishing gradient problem.

Course 4-2: Word Embeddings

Traditional Word Representation Approaches

One-Hot Encoding

So far we have been representing words by **one-hot vectors**. However, the one-hot vectors are high-dimensional (dimension of each vector is equal to the size of the vocabulary), so it is computationally expensive.

Also, one-hot vectors do not capture semantic relationships between words (the inner product between any two one-hot vectors is zero), and such representation does not generalise well to new words (one needs to increase the vocabulary but what the model has learned before tells essentially nothing about the new word).

Bag-of-Words (BoW)

Bag-of-Words is simple frequency-based representation technique in NLP. The idea is to think of each document as a bag, or a collection, of words, and then we count the **frequency of occurrence** of each word in the document. This provides a straightforward way to capture word importance, but there is loss of sequential information as the order of the words does not matter in vector representation. Also, common words like "the" or "of" tend to dominate the count vector, drowning out rarer but more informative words.

TF-IDF

TF-IDF, or Term Frequency-Inverse Document Frequency, is an improvement over the BoW representation. The idea is to increase the importance of rare words while reducing the weight of common words by considering the two factors:

- TF (term frequency): measures how frequent a particular word appears in a particular document, a word that appears many times in a document is assumed to be important to that document

$$\text{TF} = \frac{\text{frequency that the word appears in this document}}{\text{total number of words in this document}}$$

- IDF (inverse document frequency): measures the importance of a particular word across all documents, a word that appears in only a few documents would be more informative than those appear in many documents

$$\text{IDF} = \log\left(\frac{\text{total number of documents}}{\text{number of documents that contain this word} + 1}\right)$$

The **TF-IDF score** of a particular word in a particular document is given by:

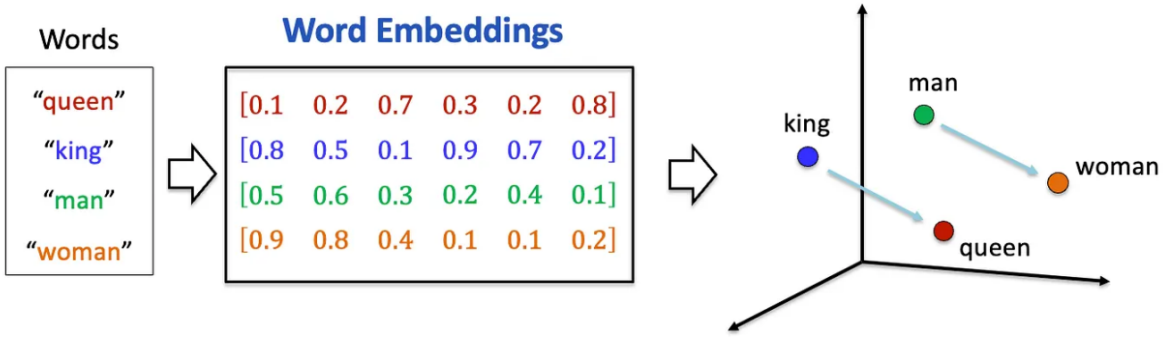
$$\text{TF-IDF} = \text{TF} \times \text{IDF}$$

For the common words which appear in almost all documents, their IDF score would be equal or close to zero, making their TF-IDF score low, which suggests that they are less important as desired. Rarer words such as technical terminologies would have higher TF-IDF scores, which can be useful in information retrieval and text analysis tasks.

Limitations of Traditional Word Representation Approaches

One major problem with traditional word representation methods is that they do not reflect any relationship between the words, so it is quite unlikely for the model to understand analogies like a man is to a woman as a boy is to a girl or a king is to a queen.

Such limitation motivated the development of **word embedding**, an technique in natural language processing that converts words into dense vectors (dimension of the vector $\ll$ size of the vocabulary), mapping semantic meaning into vectors consisting of continuous real numbers. This allows the models to better capture semantic relationships and syntactic information.



The word embeddings can be trained in an **unsupervised** manner using very large corpus and be reused for specific tasks. Suppose we want to train an RNN to recognize names in a given sentence, the model will be able to generalize more easily if we use word embeddings as inputs instead of the one-hot vectors. Even if the model sees a new word for the first time, for example say tuna, the model would know it is a fish as its word representation is similar to already seen examples like cod or salmon. Therefore, it is possible to take the learned embeddings and fine tune them for new tasks (a form of transfer learning).

Learning Word Embeddings

Similarity Functions

Given two word vectors $\mathbf{u}$ and $\mathbf{v}$, the simplest possible similarity function between two word vectors is the **Euclidean distance**:

$$\text{EuclideanDistance}(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\| = \sqrt{\sum_{i=1}^{D}(u_i - v_i)^2}$$

Another commonly-used similarity function is the **cosine similarity**:

$$\text{CosineSimilarity}(\mathbf{u}, \mathbf{v}) = \cos\theta(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}$$

where $\theta(\mathbf{u}, \mathbf{v})$ is the angle between $\mathbf{u}$ and $\mathbf{v}$. If two words are strongly correlated, then their representations would be close in the vector space, which means the angle is small so their cosine similarity is close to 1. If two words are unrelated, then due to the sparse nature of the high-dimensional vector space, their representations are almost orthogonal so their cosine similarity would be equal or very close to 0.

Embedding Matrix

An **embedding matrix** serves as a lookup table that maps one-hot word vectors to continuous word embeddings. Let's denote the embedding matrix by E, the one-hot vector for the j-th word in the vocabulary by $\mathbf{o}_j$, then the word embedding $\mathbf{e}_j$ for this word is given by:

$$\mathbf{e}_j = E\mathbf{o}_j$$

Suppose we are using a vocabulary of V words and we want to learn a d-dimensional word embedding, then the embedding matrix E is of size $d \times V$.

The task of learning a good word embedding is therefore to learn the embedding matrix E. One may initialize the embedding matrix randomly and apply gradient descent to make improvements towards optimum.

It is also worth mentioning that it is not necessary to use vector multiplication to extract word embeddings. Given the j-th one-hot vector $\mathbf{o}_j$, due to the one-hot nature (1 in the j-th row and 0's elsewhere) , the word embedding $\mathbf{e}_j = E\mathbf{o}_j$ is simply the j-th column of the embedding matrix E, so in practice we can use slicing to obtain the desired column for the embeddings instead of doing the vector multiplication.

Learning Word Embeddings: An Intuitive Approach

Suppose we want to build a language model to predict the next word in a sequence. We would first embed the input word with the embedding matrix: $\mathbf{e}_j = E\mathbf{o}_j$. The embedding is then fed into a two-layer neural network: first a hidden layer, and then a SoftMax layer to output the predicted probability over V words in the vocabulary.

The input words could be the last one word, or the last k words, and so on. The goal is to optimize the embedding matrix E and layer parameters of the neural network in order to maximize the likelihood to predict the next word given the input contexts.

Case Study: Word2vec

Let's first look at **Word2vec**, a technique to learn a meaningful word embedding in natural language processing.

Continuous Bag-of-Words (CBOW) Model

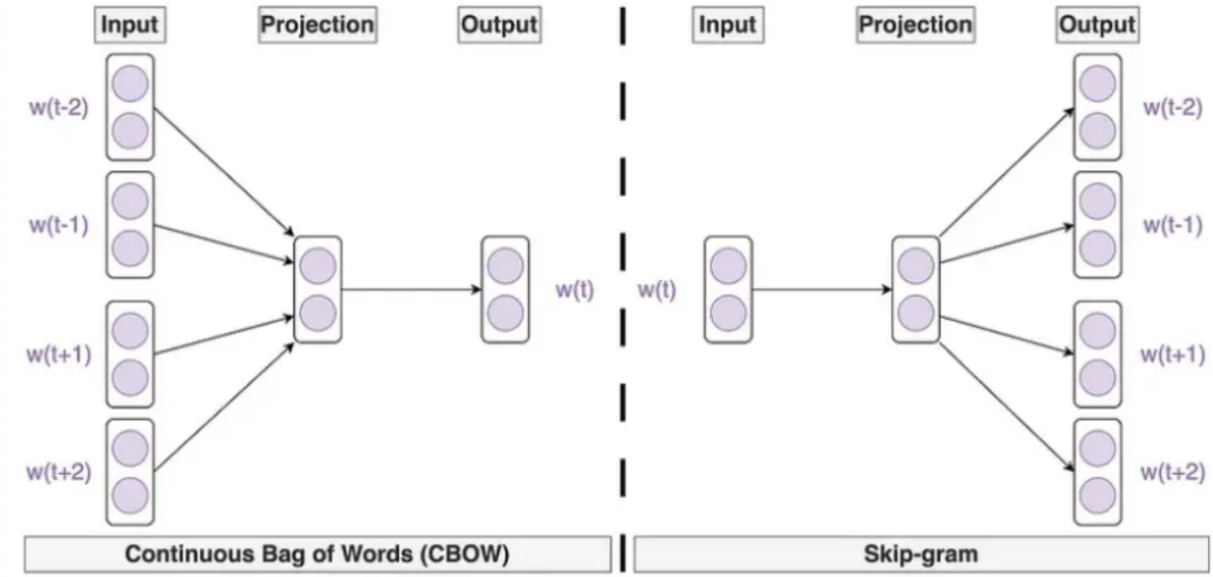
The **Continuous Bag-of-Words** model is designed to predict predict the target word based on the k context words on both sides. For example, given the sentence "I want a glass of orange juice to go along with my cereal", and if we are using a window of size $k = 2$, we would like the model to predict the target word "orange" given the context words "glass", "of", "juice" and "to".

For a desired target word `t`, the CBOW model does the following:

- compute the embeddings of each of the $2k$ context words: $\mathbf{e}_{\mathrm{c}_i} = E\mathbf{o}_{\mathrm{c}_i}$ $(i = -k, \cdots, -1, +1, \cdots, +k)$
- take average of these embeddings to give a single vector: $\bar{\mathbf{e}}_{\mathrm{c}} = \dfrac{1}{2k}\sum_i \mathbf{e}_{\mathrm{c}_i}$
- predict the target word with SoftMax: $\hat{y} = P(\mathrm{t}|\mathrm{c}) = \dfrac{\mathrm{e}^{\theta_{\mathrm{t}}^T \bar{\mathbf{e}}_{\mathrm{c}}}}{\sum_{j=1}^{V} \mathrm{e}^{\theta_j^T \bar{\mathbf{e}}_{\mathrm{c}}}}$
- use cross-entropy to evaluate the loss: $L(\hat{y}, y) = -\sum_{i=1}^{V} y_i \log \hat{y}$

Skip-Gram Model

In practice, researchers found that inverting the CBOW objective (using surrounding words to predict one target word), that is to take a single word and predict its surrounding words, yields equally meaningful word embeddings. This is the idea of the **skip-gram model**, simple but works remarkably well.



Typically, a window of fixed size (± 5 words for example) is chosen, and for a specific context word, a target word is sampled uniformly from within that window. That makes a context-target pair for training. For example, given the sentence "I want a glass of orange juice to go along with my cereal", and if we take "orange" as the context word (input), we want the model to predict target words "juice", "glass", "want", "my", "cereal", etc.

Suppose the context word is `c` and the target word is `t` so we want the model to predict `t` given `c`. The skip-gram model does the following

- compute embedding of the context word: $\mathbf{e}_\mathrm{c} = E\mathbf{o}_\mathrm{c}$
- use a SoftMax layer to predict: $\hat{y} = P(\mathrm{t}|\mathrm{c}) = \frac{\mathrm{e}^{\theta_\mathrm{t}^T \mathbf{e}_\mathrm{c}}}{\sum_{j=1}^{V} \mathrm{e}^{\theta_j^T \mathbf{e}_\mathrm{c}}}$
- use cross-entropy to evaluate the loss: $L(\hat{y}, y) = -\sum_{i=1}^{V} y_i \log \hat{y}$

Problem at the SoftMax Layer

One problem with both the CBOW model and the skip-gram model is the efficiency at the SoftMax layer. Since we need to sum over all the words in the vocabulary to evaluate the probability of each word, this could be very slow and even slower if we use bigger vocabularies.

One solution to this problem is to use a **hierarchical SoftMax classifier** which works as a tree classifier. The hierarchical SoftMax classifier uses an asymmetrical tree to reduce the computations, where common words tend to be at the top and less common words are deeper down near the bottom.

Negative Sampling

One training technique that optimizes the efficiency for training word embedding models is **negative sampling**. By learning from a positive sample and a small number of negative samples, we can convert a slow multi-classification task into a fast binary classification task.

We construct the training examples as the following. For the positive sample, we pick a context word and a nearby word as the target . For negative samples associated with the same context word, we choose the target word randomly. For each positive sample we can choose k negatives ($k = 5 \sim 20$ for smaller datasets, and $k = 2 \sim 5$ for larger datasets).

For each sample, we apply a simple logistic regression model to predict

$$\hat{y} = P(y = 1|\mathrm{c}, \mathrm{t}) = \sigma(\theta_\mathrm{t}^T \mathbf{e}_\mathrm{c})$$

So instead of having a V-dimensional classification problem (V being the vocabulary size), we only need to train $k + 1$ classifiers in each iteration. The model only needs to update a small subset of weights at a time instead of updating the weights for the entire vocabulary, so the model converges faster, making it feasible to large datasets.

Selection of Samples

If we sample randomly, we are actually taking samples according to the empirical frequency at which different words appear in a corpus, this means we would have sampled more frequent words like "the", "of", etc.

One solution to this problem is to sample according to the frequency-based probability for a given word. Let $f(w_i)$ be the frequency that the word w_i appears, then the probability of sampling this word is:

$$P(w_i) = \frac{f(w_i)^\alpha}{\sum_{j=1}^{V} f(w_j)^\alpha}$$

where $\alpha < 1$ is a hyperparameter to suppress the probability of very frequent words but raise the probability of rarer words. Andrew suggests that choosing $\alpha = \frac{3}{4}$ gives nice results.

Case Study: GloVe

GloVe, or Global Vectors for Word Representation, is another algorithm to learn a word embedding. While Word2Vec uses local context windows, GloVe uses global **co-occurrence statistics**. The main idea behind the GloVe approach is to capture semantic relationships between words by analysing the co-occurrence patterns, i.e., how often a pair of words appear together, in a large corpus.

To learn a word embedding with GloVe, we first build a co-occurrence matrix X where the matrix element X_{ij} counts the number of times that a word w_j appears in the surrounding context of word w_i (usually within a window of fixed size around the word).

Since the dot product of two word vectors reflects their correlation as in their cosine similarity, so we want to adjust the vectors so that their dot product correctly reflect the relationship between words as described by the co-occurrence matrix elements. The **GloVe cost function** to be minimized is therefore designed to be the difference between the dot product of two word vectors and the logarithm of their co-occurrence probability:

$$J = \sum_{i,j=1}^{V} f(X_{ij}) \left(e_i^T e_j + b_i + b_j' - \log X_{ij}\right)^2$$

where b_i, b_j' are bias terms for the words w_i and w_j. A specialised function $f(X_{ij})$ is introduced to ensure that word pairs like "this is" or "we can" with extremely high co-occurrence are not over-weighted. The authors of GloVe suggests the function:

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ \left(\frac{x}{x_{\max}}\right)^\alpha & \text{if } 0 < x < x_{\max} \\ 1 & \text{if } x \geq x_{\max} \end{cases}$$

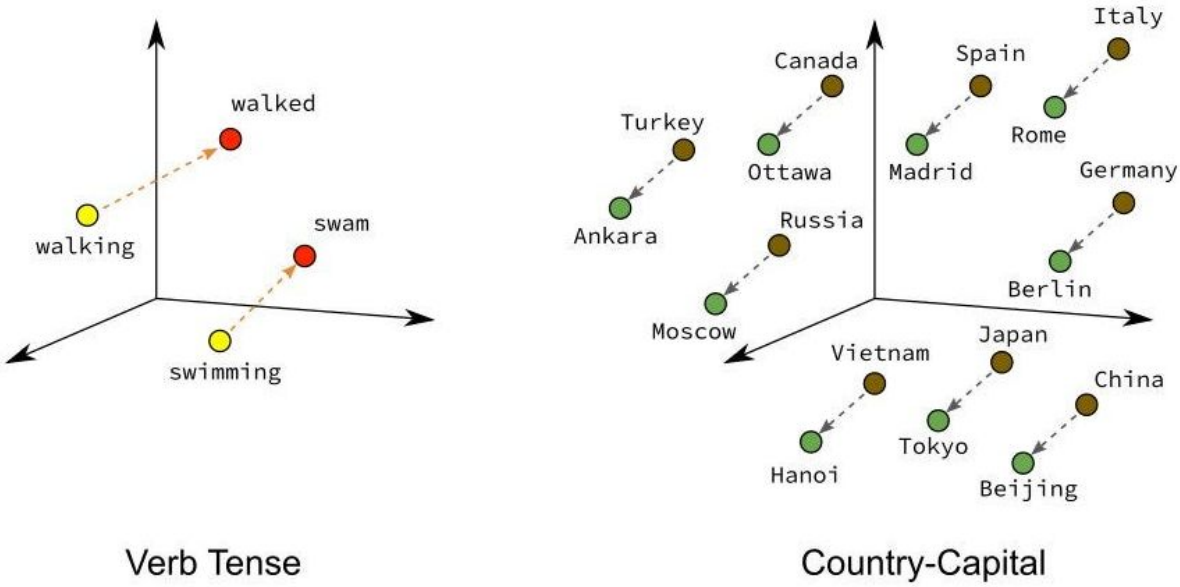
with $\alpha = \frac{3}{4}$ and $x_{\max} = 100$ determined empirically.

Note that $f(0) = 0$, this means words that do not co-occur ($X_{ij} = 0$) are simply ignored by the cost function (no need to worry about potential issue with ill-defined $\log 0$), and so GloVe trains only positive samples in contrast to Word2Vec with negative sampling.

Properties of Word Embeddings: Analogical Reasoning

One striking property of learned word embeddings is their ability to capture semantic and syntactic analogies through simple vector arithmetic. Since words that behave similarly in a corpus, i.e., words that often appear near similar neighbours, end up with similar embeddings. Consistent relationships across word pairs are encoded as consistent directions in the embedding space. Some well-known examples include:

- gender: $\mathbf{e}_{\text{king}} - \mathbf{e}_{\text{queen}} \approx \mathbf{e}_{\text{man}} - \mathbf{e}_{\text{woman}}$
- pluralisation: $\mathbf{e}_{\text{cats}} - \mathbf{e}_{\text{cat}} \approx \mathbf{e}_{\text{dogs}} - \mathbf{e}_{\text{dog}}$
- verb tenses: $\mathbf{e}_{\text{walked}} - \mathbf{e}_{\text{walk}} \approx \mathbf{e}_{\text{ran}} - \mathbf{e}_{\text{run}}$
- capital cities: $\mathbf{e}_{\text{France}} - \mathbf{e}_{\text{Paris}} \approx \mathbf{e}_{\text{China}} - \mathbf{e}_{\text{Beijing}}$



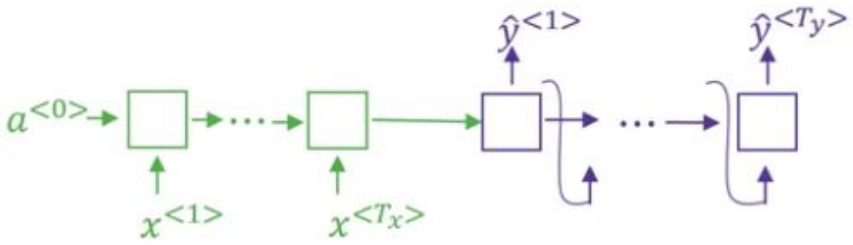
Verb Tense

Country-Capital

Course 4-3: Beam Search & BLEU Score

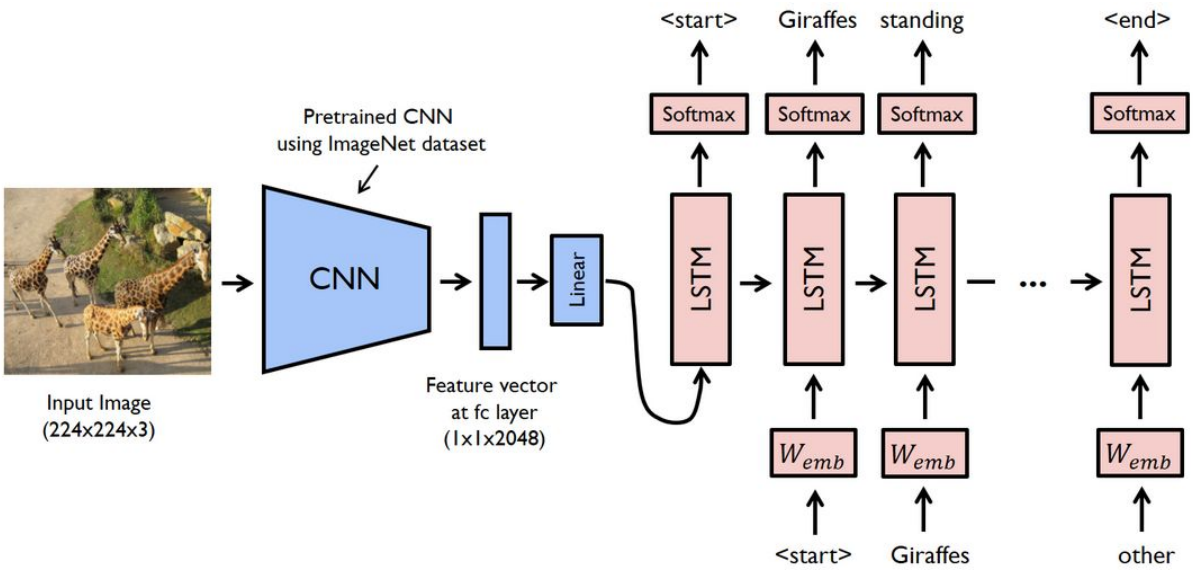
Encoder-Decoder Architecture

Let's begin with a machine translation problem, say we want to translate a French sentence X into an English sentence Y. The architecture for such sequence-to-sequence models typically include an encoder and a decoder.



- The encoder, usually an RNN (LSTM or GRU), takes the input sequence and outputs a vector that represents the whole sequence
- The decoder, also an RNN, takes the vector generated by the encoder and outputs the new sequence

A similar encoder-decoder architecture can also be used for image captioning, where a CNN plays the role as a visual encoder and the decoder is a RNN.



We have introduced a language model that can generate a sequence of words $\mathbf{y}^{\langle 1 \rangle}, \cdots, \mathbf{y}^{\langle T_y \rangle}$ based on probability maximisation. The decoder is similar to such language model, but instead of taking a zero vector as the initial hidden state $\mathbf{a}^{\langle 0 \rangle}$, the decoder takes the output vector of the encoder, so the output of the decoder is conditioned on the input sequence. Therefore, the probability that we wish to maximise would be different:

- for the language model, our objective is: $\arg \max_{\mathbf{y}^{\langle 1 \rangle}, \cdots, \mathbf{y}^{\langle T_y \rangle}} P\left(\mathbf{y}^{\langle 1 \rangle}, \cdots, \mathbf{y}^{\langle T_y \rangle}\right)$
- for the machine translation problem, our objective is: $\arg \max_{\mathbf{y}^{\langle 1 \rangle}, \cdots, \mathbf{y}^{\langle T_y \rangle}} P\left(\mathbf{y}^{\langle 1 \rangle}, \cdots, \mathbf{y}^{\langle T_y \rangle} \middle| \mathbf{x}^{\langle 1 \rangle}, \cdots, \mathbf{x}^{\langle T_x \rangle}\right)$

Beam Search

One naïve solution to find the most probable output sequence is by greedy search, where at each time step the word with the highest probability is chosen. But choosing the most probable word at the current time step does not always maximise the conditional probability as a whole, so greedy approach does not result in very good results.

A useful algorithm better than the greedy search is the **beam search**. We will talk about how beam search works in the following paragraphs.

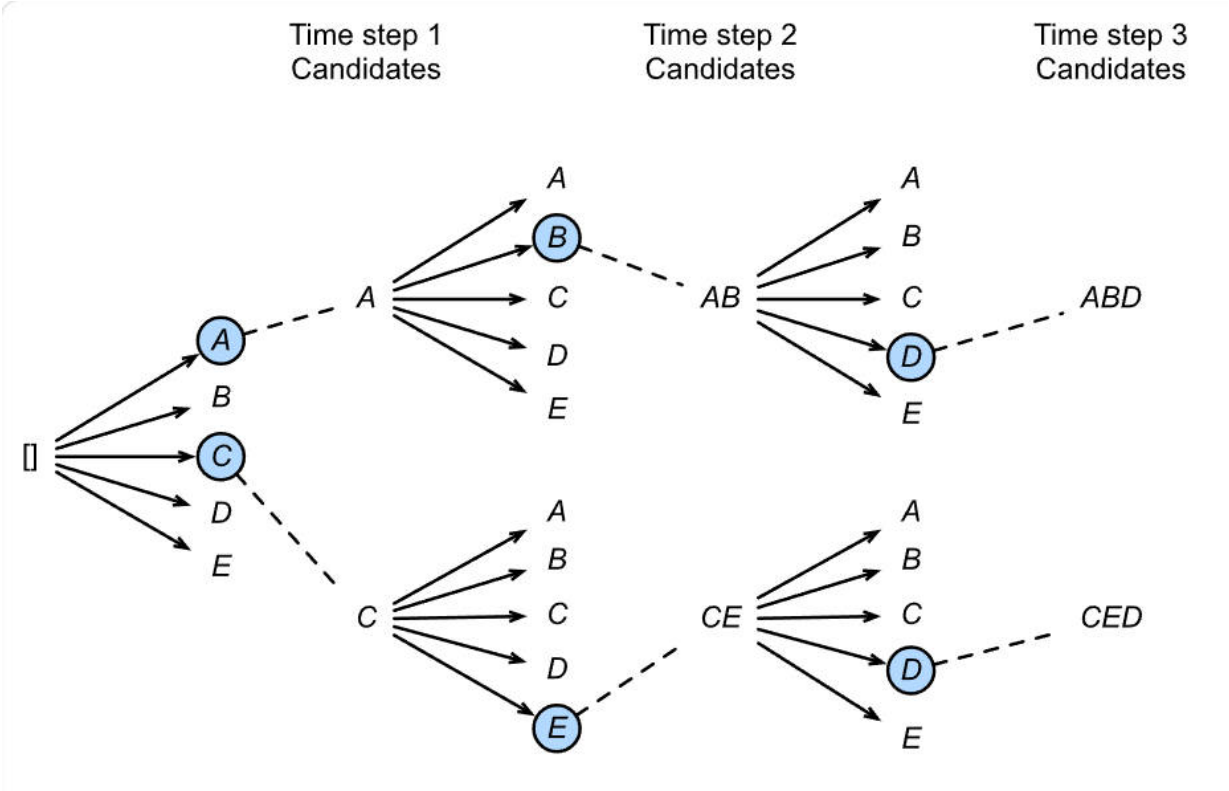
The algorithm has a parameter B called the beam width, which represents the number of the most probable words that we would keep at each time step.

Taking the machine translation problem as our example, once the decoder receives the output of the encoder, we would obtain a probability distribution over all V words for the first word of the output sentence where V is the size of the vocabulary. We then keep the top B outputs, i.e., the outputs $\mathbf{y}^{\langle 1 \rangle (1)}, \cdots, \mathbf{y}^{\langle 1 \rangle (B)}$ that have the highest probabilities $P\left(\mathbf{y}^{\langle 1 \rangle} \middle| X\right)$ among all V possible words.

For each of the $\mathbf{y}^{\langle 1 \rangle (1)}, \cdots, \mathbf{y}^{\langle 1 \rangle (B)}$, we use B copies of the same network to predict the second word $\mathbf{y}^{\langle 2 \rangle}$, so we now obtain $B \times B$ combinations of $\left(\mathbf{y}^{\langle 1 \rangle}, \mathbf{y}^{\langle 2 \rangle}\right)$. We then select the top best B combinations, i.e., the combinations with the highest probability $P\left(\mathbf{y}^{\langle 2 \rangle}, \mathbf{y}^{\langle 1 \rangle} \middle| X\right)$. Note that this probability can be calculated using:

$$P\left(\mathbf{y}^{\langle 2 \rangle}, \mathbf{y}^{\langle 1 \rangle} \middle| X\right) = P\left(\mathbf{y}^{\langle 1 \rangle} \middle| X\right) \times P\left(\mathbf{y}^{\langle 2 \rangle} \middle| X, \mathbf{y}^{\langle 1 \rangle}\right)$$

In the next step, we continue with the B doublets and then proceed to use B networks to find the most probable B triplets of words with the highest $P\left(\mathbf{y}^{\langle 3 \rangle}, \mathbf{y}^{\langle 2 \rangle}, \mathbf{y}^{\langle 1 \rangle} \middle| X\right)$, and so on.



With beam search, we keep only B instances of the network at each time step. We are not guaranteed to find the optimal solution, so it is not an exact search algorithm like BFS (breadth first search) or DFS (depth first search). If we take $B = 1$, the algorithm becomes the greedy search. Larger choices of B lead to greater chances to obtain better results but at the cost of running time. In practice, it is common to set $B \approx 10$.

Numerical Stability & Length Normalisation

Recall that we are maximising the conditional probability of the form

$$P\left(\mathbf{y}^{\langle t \rangle}, \cdots, \mathbf{y}^{\langle 1 \rangle} \middle| X\right) = P\left(\mathbf{y}^{\langle 1 \rangle} \middle| X\right) \times P\left(\mathbf{y}^{\langle 2 \rangle} \middle| X, \mathbf{y}^{\langle 1 \rangle}\right) \times \cdots \times P\left(\mathbf{y}^{\langle t \rangle} \middle| X, \mathbf{y}^{\langle 1 \rangle}, \cdots \mathbf{y}^{\langle t-1 \rangle}\right)$$

which is computed as the product of many probabilities. Since many of those probabilities are very small, so multiplying them can result in numerical overflow. One solution to this problem is to use **the sum of the logarithms of the probabilities**:

$$\arg \max_{\mathbf{y}^{\langle 1 \rangle}, \cdots, \mathbf{y}^{\langle T_y \rangle}} \sum_{t=1}^{T_y} \log P\left(\mathbf{y}^{\langle t \rangle} \middle| X, \mathbf{y}^{\langle 1 \rangle}, \cdots \mathbf{y}^{\langle t-1 \rangle}\right)$$

However, this leads to yet another problem that the optimisation function would prefer shorter sequences over longer ones. This is because probabilities are less than one, the more probabilities we multiply (or adding their logarithms, which would take negative values), the lower value we get. This leads to the **length normalisation** scheme, that is to take a biased average of the logarithms of the probabilities:

$$\arg \max_{\mathbf{y}^{\langle 1 \rangle}, \cdots, \mathbf{y}^{\langle T_y \rangle}} \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P\left(\mathbf{y}^{\langle t \rangle} \middle| X, \mathbf{y}^{\langle 1 \rangle}, \cdots \mathbf{y}^{\langle t-1 \rangle}\right)$$

where α is a heuristic parameter that controls how much we prefer shorter sentences. If $\alpha = 0$ then there is no length normalisation. If $\alpha = 1$ then there is full sequence length normalisation. In practice, somewhere in between the two extremes works best. Andrew suggests that taking $\alpha = 0.7$ is a good empirical choice.

Error Analysis in Beam Search

The error in the model predictions may be due to the beam search algorithm or the RNN model itself, so it is worth spending time on figuring out where things go wrong.

Given a predicted output $\hat{Y}$ and a possible correct translation Y^* given by a human:

- Case 1: $P\left(Y^* \middle| X\right) > P\left(\hat{Y} \middle| X\right)$

  This suggests we did not find the optimal output sequence, so beam search is at fault.

- Case 2: $P\left(Y^* \middle| X\right) < P\left(\hat{Y} \middle| X\right)$

  This means the model has assigned a higher probability to a worse translation, so RNN model is at fault as its predicted probability distribution is not accurate.

For a complete error analysis, we can take many error examples and find out what fraction of errors are due to beam search or due to RNN model. A deeper error analysis for each part can then be done based on the counts.

BLEU Score

One of the challenges in machine translation is that there could be multiple equally good translations of a given sentence. How we evaluate the accuracy of the system when there are multiple good answers is to use the BLEU score, where BLEU stands for *bilingual evaluation understudy*. For a given sentence, we collect several reference translations provided by humans. As long as the machine-generated translation is close to any of the references, it gets a high BLEU score.

To begin with, we may evaluate the machine-generated output is to look at each word in the output and check whether it is in any one of the references. This is called the **precision**. However, this is not a useful measure because the machine can output a list of repeated common words (like "The the the the the the.") to gain very high precision. So it is better to use a modified precision: for a particular word, we look for the maximum number of occurences of this word in the reference sentences, and clip the count by this maximum number.

But a bunch of matching words does not necessarily mean that the candidate is a good translation, as the words might come out in a random order so that the sentence is not well structured. We can further evaluate the quality of the machine-generated output by looking at the overlap of more than one words at a time, or n-grams, as they can capture local word order and phrasal coherence. The modified precision of the n-gram is:

$$p_n = \frac{\sum_{n\text{-gram} \in \hat{Y}} \text{count\_clip\_of\_}n\text{-gram}}{\text{number\_of\_candidate\_}n\text{-grams}}$$

Let's illustrate this with the same example in Andrew's course.

- Original French sentence X: Le chat est sur la tapis.
- Human reference $Y1$: The cat is on the mat.
- Human reference $Y2$: There is a cat on the mat.
- Machine output $\hat{Y}$: The cat the cat on the mat.

If we look at n-grams at order 2, i.e., bigrams, we construct a table like this:

2-grams	Count	Count clip
the cat	2	1 ($Y1$)
cat the	1	0
cat on	1	1 ($Y2$)
on the	1	1 ($Y1$ or $Y2$)
the mat	1	1 ($Y1$ or $Y2$)
Total	6	4

The precision score for the bigram in this example is therefore: $p_2 = \frac{4}{6}$.

The **combined BLEU score** for n-grams of different sizes up to some maximum order N (usually up to 4) is:

$$\text{BLEU} = \text{BP} \times \exp\left(\frac{1}{N}\sum_{n=1}^{N} \ln p_n\right)$$

The term BP is the **brevity penalty** factor, which deals with the case where a model gains a good precision score by outputting fewer words while missing some bits of information. Let c be the length of the machine-generated candidate and r be the length of closest reference translations (some take r as the average length of reference translations), the BP penalty factor is given by:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r \end{cases}$$

For very short output sentence, $\frac{r}{c} > 1$, so $\exp\left(1 - \frac{r}{c}\right) < 1$, this means short translations are indeed penalised.

BLEU score was proposed by researchers at IBM T. J. Watson Research Center in 2001 so that the evaluation of machine translations can be quicker, cheaper and less human labour. It remains a popular automated and inexpensive metrics in machine translation and has many open source implementations. The known limitations of BLEU is that it does not account for synonyms and fluency, and BLEU has been partially superseded by metrics like METEOR and BERTScore in modern NLP researches.

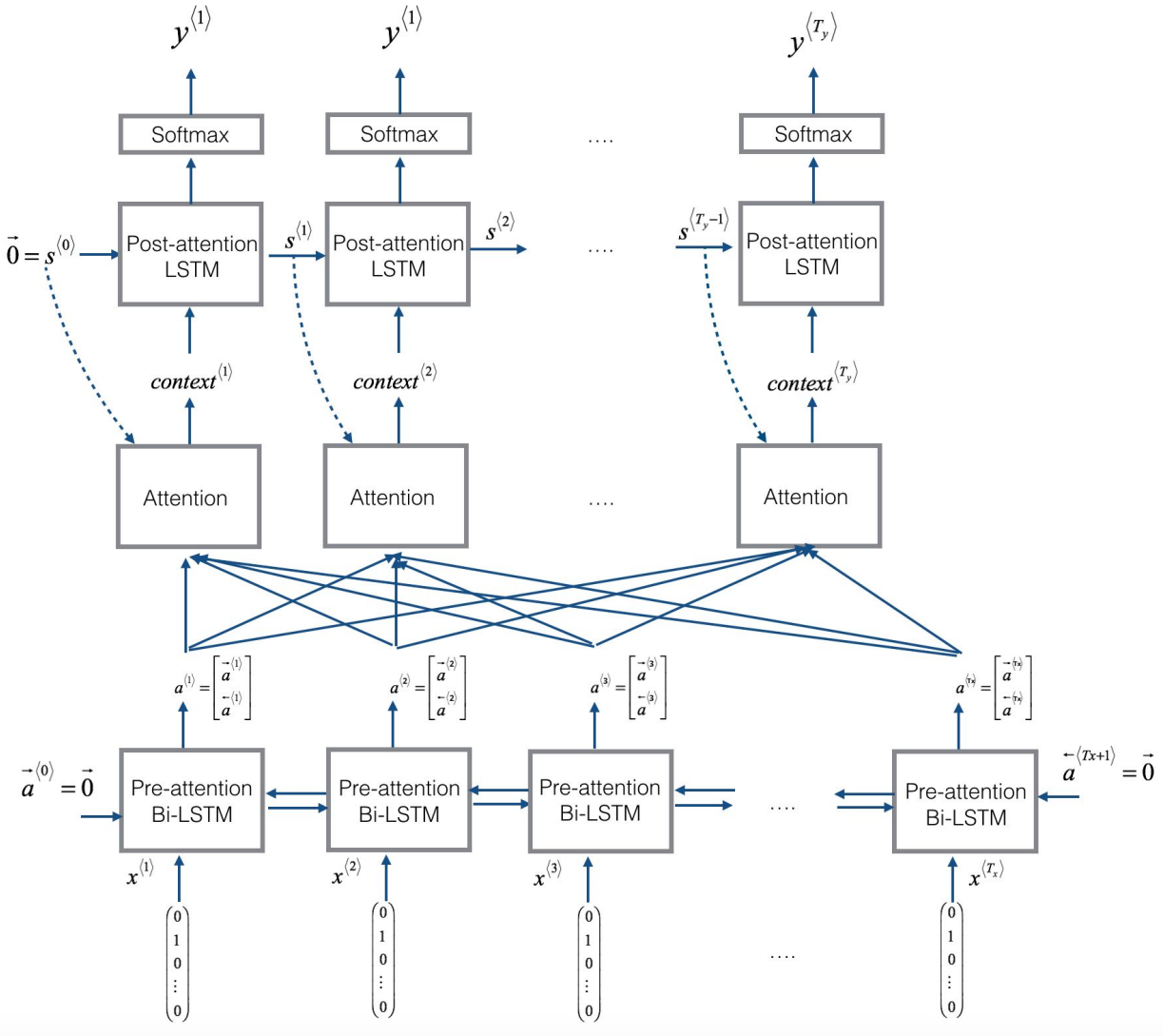
Course 4-4: Attention Mechanism & Transformer

Attention Mechanism

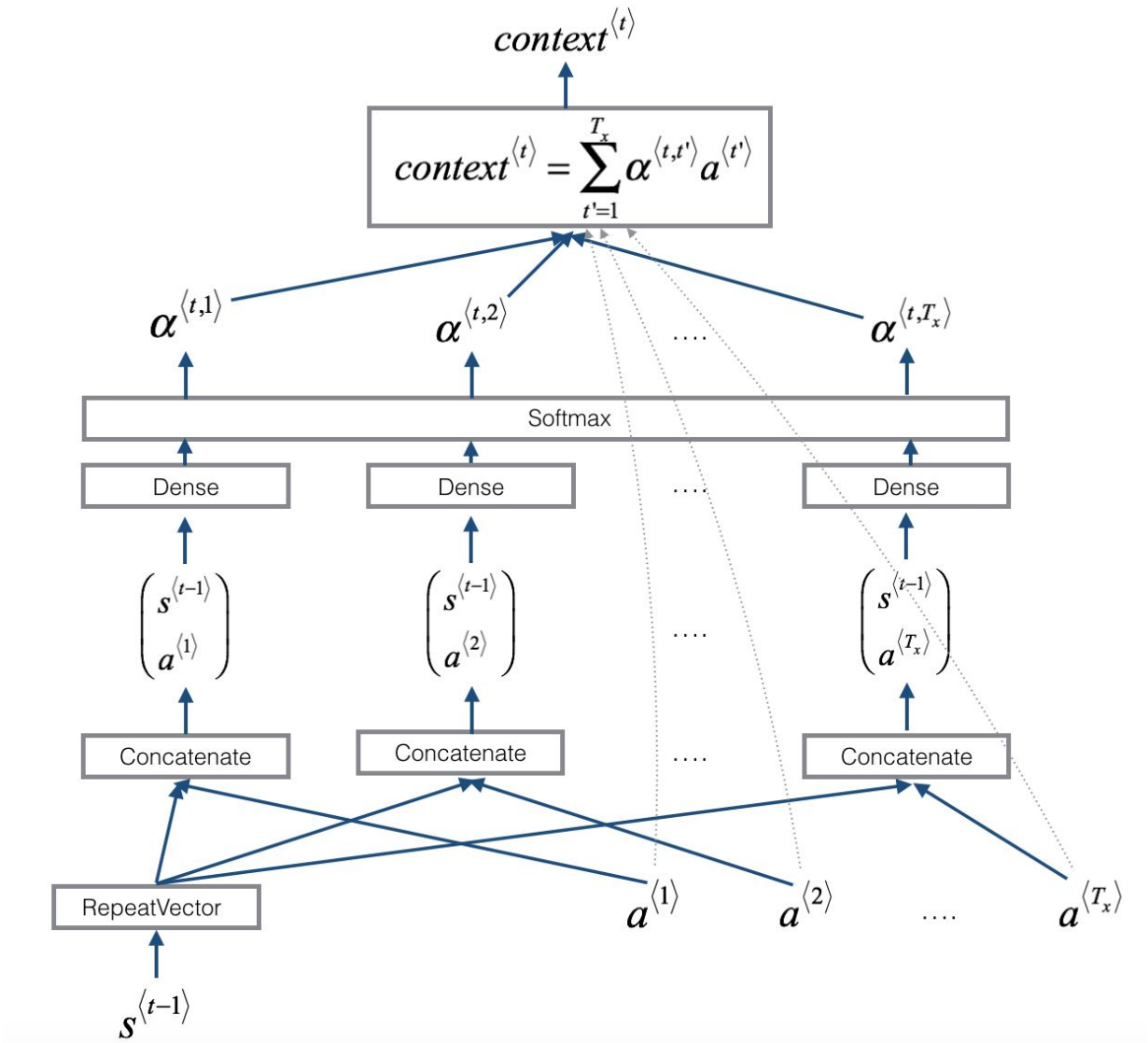
We have introduced the encoder-decoder architecture for tasks like machine translation, and it works quite well with short sentences. But researchers found that as the length of the sequence increases, model performance metrics like the BLEU score decreases significantly. This motivates a modification of such architecture called the attention model.

The idea of the **attention mechanism** is to use the output vectors of all the words in the sequence instead of just using the output vector of the last word. At a particular time step t, the network may determine which words are important for this output position and only use the information from those words, i.e., pay attention to only those parts of the input sequence, to predict the output vector.

The way to do this is to first take all the activations from a pre-attention RNN corresponding to the input sequence, and aggregate them into a single context vector of fixed length. At each time step, we feed this context vector into a post-attention RNN to predict the output vector. This is illustrated by the diagram shown below.



The next diagram illustrates with further details how each one of the vectors $\text{context}^{\langle t \rangle}$ is computed:



The **one-step attention context** is computed as a weighted sum of the pre-attention activations:

$$\text{context}^{\langle t \rangle} = \sum_{t'=1}^{T_x} \alpha^{\langle t,t' \rangle} \mathbf{a}^{\langle t' \rangle}$$

where in this formula:

- $\alpha^{\langle t,t' \rangle}$ are the **attention weights** representing the amount of attention that $\hat{y}^{\langle t \rangle}$ should be paying to the activation $\mathbf{a}^{\langle t' \rangle}$
- each activation $\mathbf{a}^{\langle t' \rangle}$ corresponds to a specific input word can contain a forward component and a backward component if we use a bidirectional pre-attention RNN, that is: $\mathbf{a}^{\langle t \rangle} = [\overrightarrow{\mathbf{a}}^{\langle t \rangle}, \overleftarrow{\mathbf{a}}^{\langle t \rangle}]$

The attention weights are computed with a SoftMax layer as:

$$\alpha^{\langle t,t' \rangle} = \frac{\mathrm{e}^{e^{\langle t,t' \rangle}}}{\sum_{t'=1}^{T_x} \mathrm{e}^{e^{\langle t,t' \rangle}}}$$

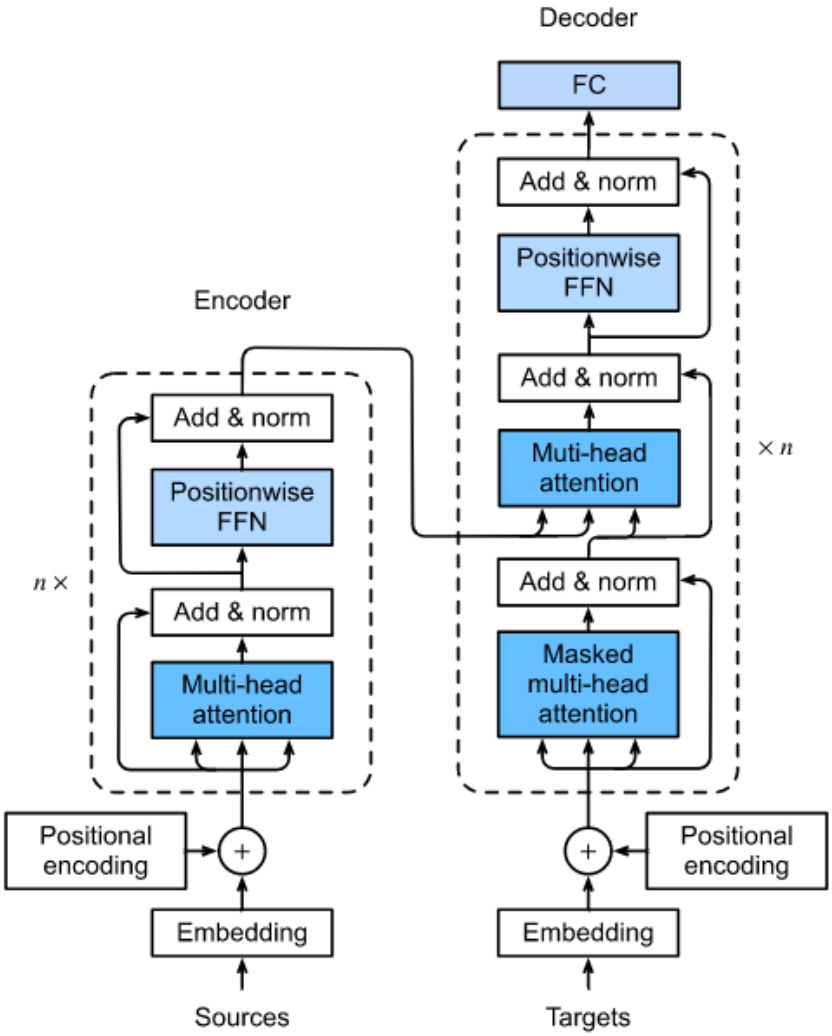
where $e^{\langle t,t' \rangle}$ are the **energies**, which are computed with a simple neural network (typically a small feed forward network, sometimes just a single-layer with a $\tanh$ function as the activation). This network takes the hidden state of the post-attention RNN from the previous time step, $\mathbf{s}^{\langle t-1 \rangle}$ (also called a repeat vector), together with the current activations of the pre-attention RNN, to output $e^{\langle t,t' \rangle}$. Note that the computations for the energy variables are not explicitly shown.

The Transformer Architecture: A First Look

Fundamentally, the machine translation models that we have been studying so far operate on the principle of sequential computation, i.e., the computation of the most probable next word at each time step in these traditional RNN depend on the previous computations. As a consequence, such models do not well handle very **long-range dependencies** as early tokens must propagate through many hidden states without degrading before they could influence later outputs. Also, the sequential nature of traditional RNN makes **parallelization** essentially impossible.

Stemmed from these limitations, the transformer architecture was designed to solve such sequential tasks more efficiently and more effectively by using a **self-attention mechanism**. Transformer was first introduced in the famous paper "*Attention is All You Need*" in 2017 and has revolutionized the approach to text-generative models. Transformers are also applied in image recognition tasks, protein structure predictions, audio generation and many other domains. The use of self-attention paired with traditional convolutional networks allows for parallelization which greatly speeds up training.

The overall architecture of the **Transformer** is shown below (Picture Credit: *Dive into Deep Learning* by A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola).



The transformer model consists of two main blocks: the **encoder** and the **decoder**.

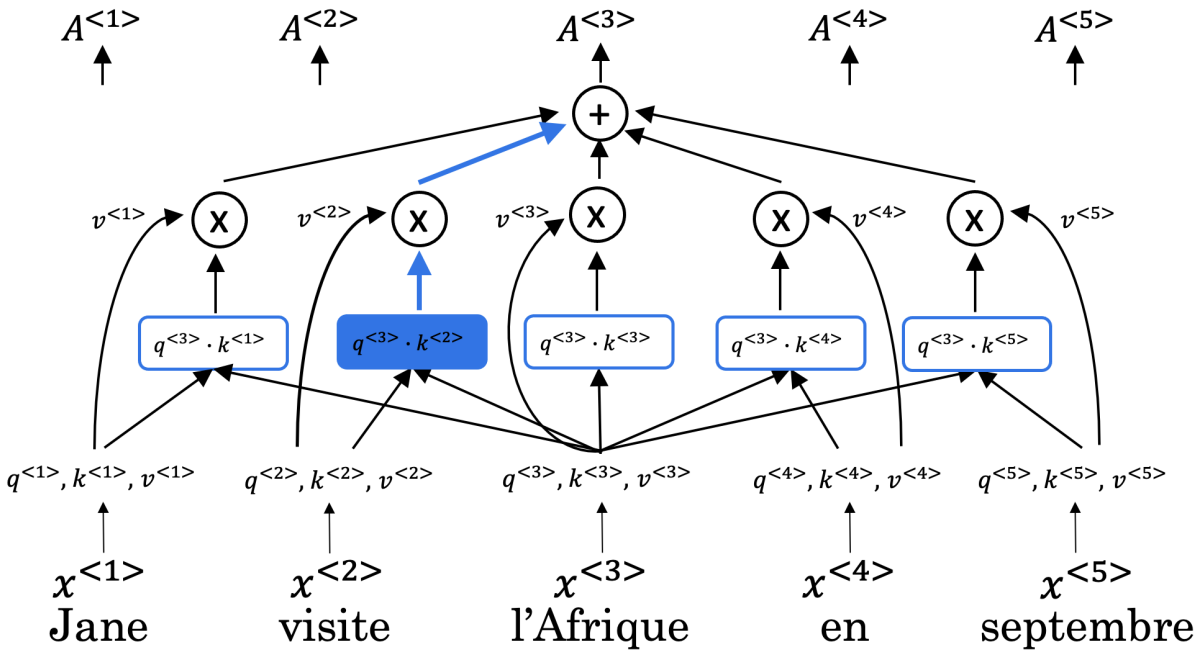
The encoder is responsible for making sense of the input. It reads the input sequence and creates a representation of the input that contains a complete understanding of the meaning of the input sequence. Lying at the core of the encoder is the **multi-head self-attention mechanism**, which allows every word to interact with every other word in the input, so that relationships and dependencies are understood.

The decoder then generates the output sequence, one word at a time. The decoder uses a **masked self-attention mechanism**, so that it only looks at the previously generated outputs but not the future ones, and then uses a **multi-head cross attention** to focus on relevant parts of the encoder's representation to generate the next output.

There is yet one further important ingredient of the architecture, **positional encoding**, which allows the model to keep track of the relative order of words in the input sequence. Although positional encoding is applied at early stages for both encoder and decoder, we would talk about that later. Let's first look at the most important mechanism of the transformer architecture, self-attention.

Self-Attention

The self-attention is basically a **scaled dot product attention**, which takes in a **query** $\mathbf{q}$, a **key** $\mathbf{k}$ and a **value** $\mathbf{v}$ as inputs to return attention-based vector representations of the words in the input sequence.



This can be mathematically expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- Q is the matrix of **queries**
- K is the matrix of **keys**
- V is the matrix of **values**
- d_k is the dimension of the keys, and division by $\sqrt{d_k}$ is used to scale everything down so that the SoftMax does not explode

Intuitively, we can think of the role of $\mathbf{q}$, the query, as asking the question: for the current input word, what is the most relevant thing to search for? Then we check that with $\mathbf{k}$'s, the keys of every other word, to see which words are worth paying more attention to (greater values of the dot product $\mathbf{q} \cdot \mathbf{k}$ means stronger relationship). Such words would return a vector $\mathbf{v}$ with which we can finally take a weighted sum to compute the attention vector.

This process is very much like how we find a desired book in a library. Suppose we want to get a book on hardcore science so we ask whether there is anything to read on theoretical physics (that is the query). We then skim through the catalogue with the book titles and introductions like *The Lord of the Rings*, *Hamlet* or *Quantum Field Theory* (these are the keys), and we find that *Quantum Field Theory* is the best match, so we get that textbook and read the contents (that is the value).

Note that with self-attention, all positions are processed simultaneously, enabling parallelization over the input sequence. Also, any two positions in the sequence can interact directly regardless of how far apart they are, whereas traditional RNNs require multiple sequential steps. That allows the transformer model to retain long-range dependencies better.

Multi-Head Attention

Multi-head attention extends the self-attention mechanism by applying self-attentions for multiple times in parallel, so that each head can focus on different aspects of the input data. This allows the model to capture more complex relationships between words from different perspectives. The outputs of the multiple heads are then concatenated to produce the output.

Mathematically, multi-head attention is computed as a linear transformation of the concatenated representation via learnable parameters W_o by:

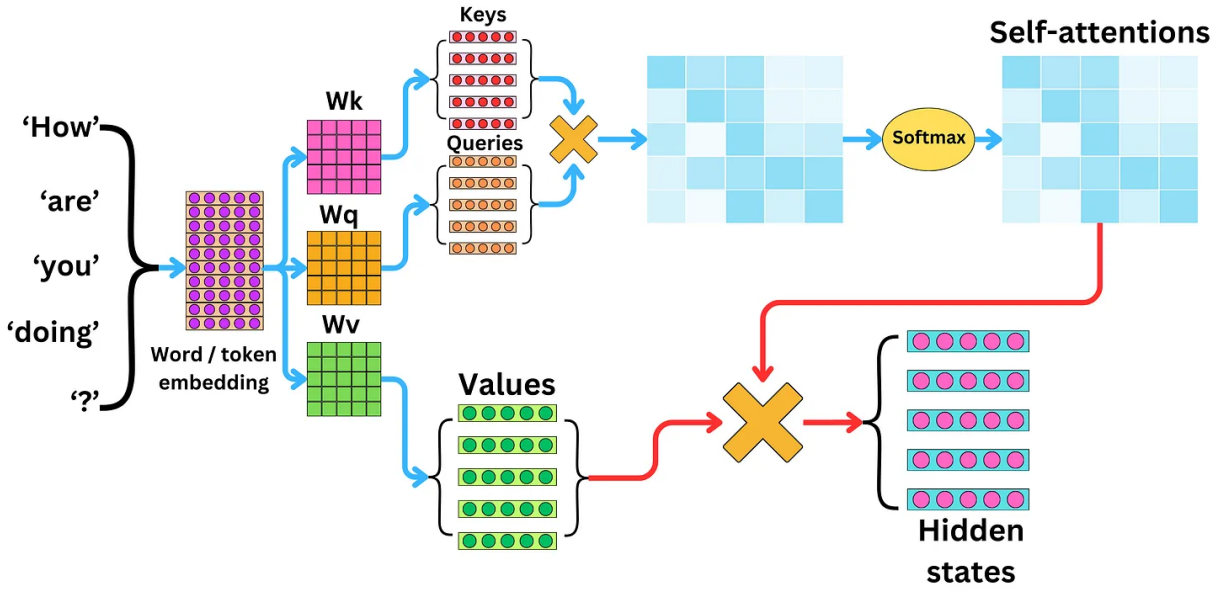
$$\text{MultiHead}(Q, K, V) = \text{Concat}\left(\text{head}_1, \text{head}_2, \cdots, \text{head}_h\right) W_o$$

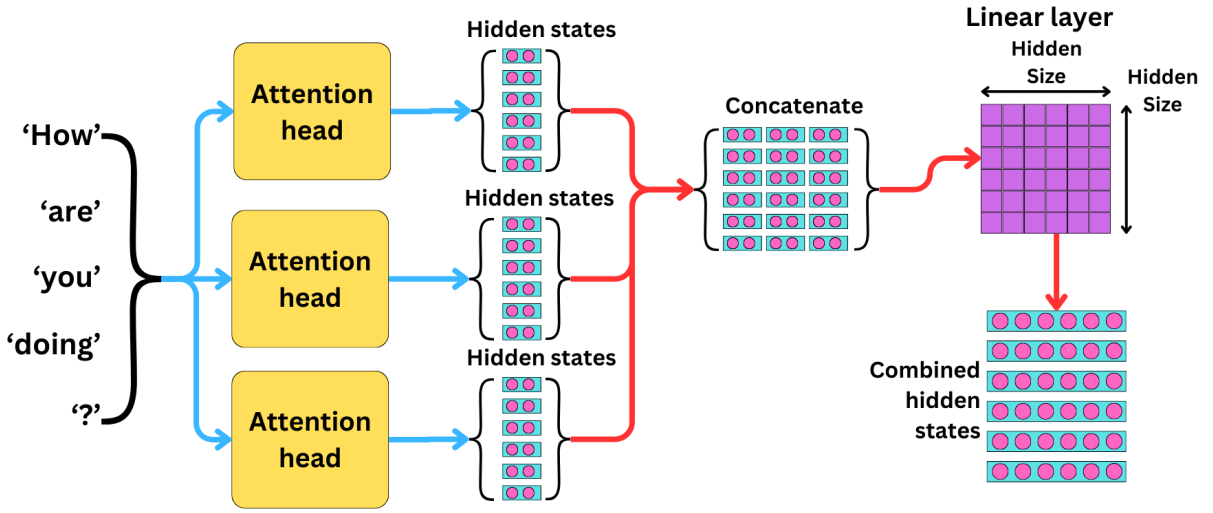
where attention vector of each head is:

$$\text{head}_i = \text{Attention}(W_i{}^Q Q, W_i{}^K K, W_i{}^V V)$$

Here, $W_i{}^Q$, $W_i{}^K$ and $W_i{}^V$ are learnable matrices for the i-th head. These projection matrices are introduced to reduce the dimensionality of the hidden states for faster training.

The following diagrams beautifully summarise the computation of a single-head self-attention and the multi-head self-attention (Picture Credit: *The AiEdge Newsletter* by Damien Benveniste).





Masked Self-Attention

There are two types of useful masks twhen building the Transformer network: the **padding mask** and the **look-ahead mask**.

To have uniform length for input sequences, those sequences longer than the maximum length would be truncated and zeroes are added to sequences shorter than the maximum length. However, these zeros will affect the SoftMax calculation. This is when a padding mask comes in handy. One approach is to define a mask that sets all the zeros in the sequence to a value close to negative infinity, so that the zeroes do not affect the score when we take the SoftMax.

The look-ahead mask follows a similar intuition. In training, we have access to the entire correct output sequence of the training example. The look-ahead mask prevents the model from cheating by looking ahead, ensuring it only makes prediction on the next output based on past and current tokens. This can be done similarly by masking out the future tokens with values close to negative infinity.

Positional Encoding

While self-attention mechanism is able to capture relationships between words from a global perspective, it does not preserve the relative order of the words which are also extremely important in sequential tasks.

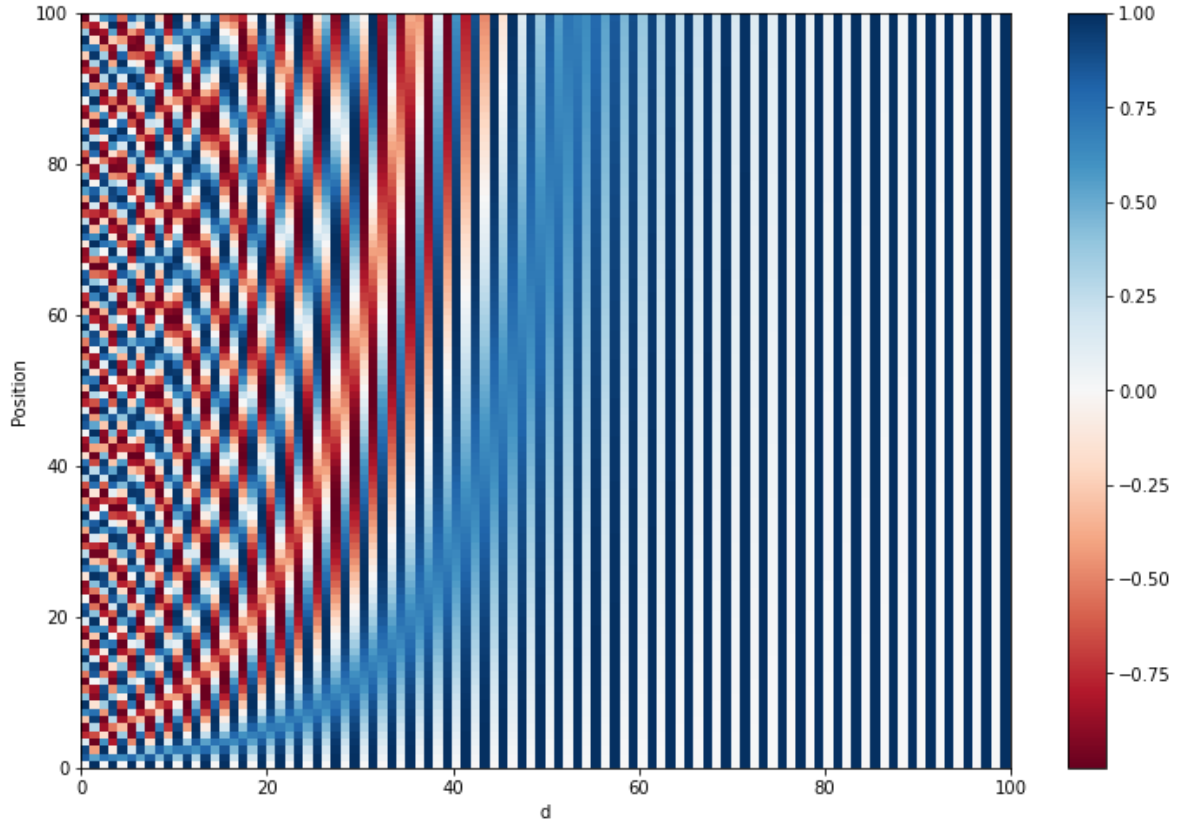
Positional encoding is a standard approach to reintroduce positional information into embedding vectors. The sum of the positional encoding and word embedding is ultimately what is fed into the model. Here are the positional encoding equations we use for the transformer architecture:

$$\mathrm{PE}(p,k) = \begin{cases} \mathrm{PE}(p,2i) = \sin\left(\dfrac{p}{10000^{\frac{2i}{d}}}\right) & \text{if } k = 2i \\ \mathrm{PE}(p,2i+1) = \cos\left(\dfrac{p}{10000^{\frac{2i}{d}}}\right) & \text{if } k = 2i+1 \end{cases}$$

where

- p is the position of the word in the sequence (i.e., the first word in the sentence has $p = 0$, the second word has $p = 1$, and so on)
- k refer to the component of the positional encoding vector with $i = k//2$, and by convention we use sine for even components and cosine for odd components
- d is the dimension of the word embedding and positional encoding

The following plot shows the heatmap for positional encodings with $d = 100$.



To develop some intuition about positional encodings, we can think of them broadly as a feature that contains the information about the relative positions of words. Word embeddings get enriched with positional information with the positional encoding being added in, but they are not significantly distorted as the values of the sines and cosines are small numbers.

Notice some interesting properties of the matrix. The first is that the norm of each of the vectors is always a constant. Let's assume that d is even, then for any value of p, we have

$$\begin{aligned}
\|\mathrm{PE}(p)\|^2 &= \sum_{k=0}^{d-1} \left(\mathrm{PE}(p,k)\right)^2 \\
&= \sum_{i=0}^{\frac{d}{2}-1} \left[\sin^2\left(\frac{p}{10000^{\frac{2i}{d}}}\right) + \cos^2\left(\frac{p}{10000^{\frac{2i}{d}}}\right)\right] \\
\Rightarrow \|\mathrm{PE}(p)\|^2 &= \frac{d}{2}
\end{aligned}$$

which shows that the norm of the positional encoding would always be the same value. Hence, the dot product of two positional encoding vectors is not affected by the scale of the vector, which has important implications for correlation calculations.

We can also consider the norm of the difference between two positional encoding vectors separated by r positions is also constant if we keep r fixed:

$$\begin{aligned}
\|\mathrm{PE}(p+r) - \mathrm{PE}(p)\|^2 &= \sum_{k=0}^{d-1} \left(\mathrm{PE}(p+r,k) - \mathrm{PE}(p,k)\right)^2 \\
&= \sum_{i=0}^{\frac{d}{2}-1} \left\{\left[\sin\left(\frac{p+r}{10000^{\frac{2i}{d}}}\right) - \sin\left(\frac{p}{10000^{\frac{2i}{d}}}\right)\right]^2 + \left[\cos\left(\frac{p+r}{10000^{\frac{2i}{d}}}\right) - \cos\left(\frac{p}{10000^{\frac{2i}{d}}}\right)\right]^2\right\} \\
&= \sum_{i=0}^{\frac{d}{2}-1} \left\{2 - 2\sin\left(\frac{p+r}{10000^{\frac{2i}{d}}}\right)\sin\left(\frac{p}{10000^{\frac{2i}{d}}}\right) - 2\cos\left(\frac{p+r}{10000^{\frac{2i}{d}}}\right)\cos\left(\frac{p}{10000^{\frac{2i}{d}}}\right)\right\} \\
&= \sum_{i=0}^{\frac{d}{2}-1} \left\{2 - 2\cos\left(\frac{p+r}{10000^{\frac{2i}{d}}} - \frac{p}{10000^{\frac{2i}{d}}}\right)\right\} \\
&= \sum_{i=0}^{\frac{d}{2}-1} \left\{2 - 2\cos\left(\frac{r}{10000^{\frac{2i}{d}}}\right)\right\} \\
\Rightarrow \|\mathrm{PE}(p+r) - \mathrm{PE}(p)\|^2 &= d - 2\sum_{i=0}^{\frac{d}{2}-1} \cos\left(\frac{r}{10000^{\frac{2i}{d}}}\right)
\end{aligned}$$

This shows that the difference between two positional encoding vectors separated by a fixed distance has a norm does not depend on the absolute positions of each encoding, but depends on the relative separation only. We can further show that we can shift the position p by any fixed offset r with a linear transformation (something very similar to a rotational matrix) that is independent of p, i.e., it is possible to express the positional encoding at position $p + r$ as linear function of the positional encoding at position p. This makes it easier for the self-attention mechanism to learn and rely on relative distances, such as two words before or three words after.

Other Crucial Components in the Transformer Model

Feed Forward Network (FFN)

There is another important component located after the multi-head attention within both the encoder and decoder blocks known as the **feed forward network (FFN)**. It acts as a position-wise processing unit that refines the representations after attention. By applying ReLu or GELU as activation functions, FFN introduces **non-linearity** into the model so complex non-linear relationships can be learned. FFN is also **position-wise independent**, meaning that the same transformation is applied to every token independently, so the computations can be done in parallel, which makes the model highly efficient.

Residual Connections and Layer Normalization

Each layer within the encoder and decoder blocks - whether a multi-head attention layer or a feed forward network - is wrapped with two additional supporting operations: a **residual connection** followed by **layer normalization**.

The residual connection adds the sub-layer's input directly to its output before passing the result forward. This technique was originally introduced in ResNets for image recognition to address the vanishing gradient problem, and the transformer architecture borrows it directly. These shortcut connection highways allow gradients to flow directly across layers, mitigating vanishing gradients and enabling more stable convergence.

Layer normalization then normalizes the summed output across the feature dimension (zero mean and unit variance). This also stabilizes the hidden state activations throughout the network, preventing them from growing too large or collapsing to zero as the signal passes through many layers.

Together, residual connections and layer normalization make it practical to stack many encoder and decoder layers without the training process becoming unstable.

Transformers: Wrapping Up

Let's finish this note by revisiting the transformer architecture.

- First the input sequence passes through an encoder that contains repeated encoder layers of
    - embedding and positional encoding of the input
    - multi-head attention on the input
    - feed forward neural network to help detect features
- Then the predicted output passes through a decoder, consisting of the decoder layers of
    - embedding and positional encoding of the output
    - masked multi-head attention on the generated output
    - multi-head cross attention with the Q from the masked multi-head attention layer and the K and V from the encoder
    - a feed forward neural network to help detect features
- Finally, after the decoder layers, one dense layer and a SoftMax are applied to generate prediction for the next output in the sequence.