

APS 360: Introduction to Artificial Intelligence

Lynne Liu | klin.liu@mail.utoronto.ca | APS360 Final Review Package

The notes are in backward order; the latest is the first.

Transformers.....	8
General.....	8
1. Quick Recap.....	8
Attention Mechanism.....	8
2. Attention Mechanism.....	8
Transformers.....	8
3. Transformers.....	8
4. Attention in transformers.....	9
5. Multi-head attention.....	9
6. Transformer encoders.....	9
7. Positional Encoding.....	9
8. Pytorch implementation.....	10
Transformers for Language Modeling.....	10
9. Language Modeling.....	10
10. BERT (Bidirectional Encoder Representations from Transformers).....	10
11. Input Embeddings.....	10
12. Task 1: Masked Word Prediction.....	10
13. Task 2: Next Sentence Prediction.....	11
14. Transfer Learning.....	11
Transformers for Computer Vision.....	11
15. ViT (Vision Transformers).....	11
Graphical Neural Networks (GNN).....	12
1. Motivation.....	12
Deep Sets.....	12
2. Sets.....	12
3. Deep Sets.....	12
Graphs.....	12
4. Transformers without positional encoding.....	12
5. Graphs in general.....	12
GNN.....	12
6. Predict links between nodes.....	13
7. Message-Passing.....	13
8. Read-out (graph pooling) function.....	14
9. GNN models.....	14
10. Graphical convolutional networks (GCNs).....	14
11. Deeper with GNNs.....	14
12. Graph Attention Networks (GAT).....	15
Pytorch Implementation.....	15
13. Dense Implementation.....	15
14. Sparse Implementation.....	15
15. DataLoader & Dataset.....	16
Generative Adversarial Networks.....	18
Generative Models.....	18
1. Identify different models.....	18
2. Generative learning.....	18

3. Generative models.....	18
4. Problem with autoencoders.....	18
Generative Adversarial Networks.....	18
5. Generative Adversarial Networks.....	18
6. GAN model.....	18
7. Pytorch implementation.....	19
Problems of Training GANs.....	20
8. Vanishing Gradients.....	20
9. Mode Collapse.....	20
10. Failing to Converge.....	20
Applications of GANs.....	20
11. Grayscale to Color.....	20
12. Conditional Generation.....	20
13. Style Transfer.....	20
Adversarial Attacks.....	20
14. Adversarial Attacks.....	21
15. Targeted/Non-targeted Attack.....	21
16. White-Box/Black-Box Attacks.....	21
17. Applications.....	21
18. Defense Against Adversarial Attack.....	21
Recurrent Neural Network.....	22
Motivation.....	22
1. Autoencoders:.....	22
• used to learn an embedding space.....	22
2. Numerical Features.....	22
3. One-hot encoding.....	22
Word Embeddings.....	22
4. Characteristics.....	22
5. Text as Sequence.....	22
6. word2vec.....	22
7. Skip-Gram Model.....	23
8. CBOW (Continuous Bag of Words) Model.....	23
9. CBOW versus Skip-Gram.....	23
10. GloVe: Utilizes the global statistics.....	24
11. PyTorch GloVe Embeddings.....	24
Distance Measures.....	24
12. Measure of distance in the embedding space.....	24
13. Word Analogies.....	24
14. Bias in Word Embeddings.....	24
Language Models.....	24
15. Language Modelling.....	24
16. Working with Text.....	25
17. Sentiment Analysis.....	25
18. Limitations.....	25
Recurrent Neural Networks (RNNs).....	26
19. RNNs.....	26
20. RNN Layers.....	26
21. PyTorch Implementation.....	26

22. Sequential Learning.....	27
23. Different RNN models.....	27
24. Different types of prediction.....	28
Limitations of Vanilla RNNs.....	28
25. Problem.....	28
26. Exploding/vanishing gradients.....	28
LSTMs & GRUs.....	28
27. Gating Mechanism.....	29
28. Long Short-Term Memory (LSTM).....	29
29. Different gates for LSTM.....	29
30. Gated Recurrent Unit (GRU).....	30
31. LSTM/GRU versus RNN.....	30
32. PyTorch implementation.....	30
Deep & Bidirectional RNNs.....	31
33. Bidirectional RNNs.....	31
34. Deep RNNs.....	31
35. PyTorch implementations.....	31
Sequence-to-Sequence Models.....	32
36. RNN Model Types.....	32
37. Hidden State Differences.....	32
38. Sequence-to-Sequence RNNs.....	32
39. During training.....	33
40. Teacher forcing.....	33
41. During Inference.....	33
42. PyTorch implementation.....	34
Unsupervised Learning.....	35
Motivation.....	35
1. Challenges with Supervised Learning.....	35
2. Feature Clustering:.....	35
• Learn the underlying patterns, then need a few examples just to label.....	35
3. Definitions.....	35
Autoencoders.....	35
4. General information.....	35
5. Applications.....	35
6. PyTorch implementations.....	36
7. Stacked Autoencoders.....	36
8. Denoising Autoencoders.....	36
9. Generating New Images with Interpolation.....	36
Variational AutoEncoders (VAE).....	37
10. Characteristics.....	37
11. Different types autoencoders.....	38
Convolutional autoencoder.....	38
12. Convolutional autoencoder.....	38
13. Transposed Convolution.....	38
14. Padding.....	38
15. Strides.....	38
16. PyTorch implementations.....	38
Pre-training with Autoencoders.....	40

17. Pre-training.....	40
Self-Supervised Learning.....	40
18. Self-supervised learning with pretext tasks.....	40
19. RotNet.....	40
20. Contrastive Learning.....	40
21. SimCLR.....	41
Convolutional Neural Network.....	42
Motivation.....	42
1. Inductive reasoning:.....	42
• Start with an observation, leads to a possible generalization hypothesis. Valid observation may lead to different hypotheses, some of them can be false.....	42
2. Inductive bias.....	42
3. Downsides for using a large fully connected network.....	42
Convolution Operator.....	42
4. Convolution.....	42
5. Convolution in 2D for images.....	42
Convolutional Neural Networks.....	43
6. Biological Influence.....	43
7. Detecting:.....	43
• The output (activation) is high if the feature is present.....	43
8. Feature:.....	43
• something in the image, like an edge, blob, or shape.....	43
9. Convolutions with learned kernels:.....	43
• share the same parameters across different locations (assuming input is stationary).....	43
10. Characteristics.....	43
11. CNNs.....	43
12. Forward and Backward pass.....	44
13. Zero Padding.....	44
14. Stride.....	44
15. Computing the output size.....	44
16. Convolutional Neural Networks (ConvNets or CNNs).....	44
17. CNN on RGB.....	44
18. Convolution on RGB input.....	44
19. Detect multiple features.....	45
20. Convolution on RGB input example.....	45
Pooling Operator (something that reduces resolution).....	45
21. Consolidating information.....	45
22. Max pooling (High-pass filter).....	45
23. Average pooling:.....	46
• Compute the average value as the selected value.....	46
24. Stride convolution:.....	46
• Shift the kernel by s (e.g. $s = 2$) when computing convolution.....	46
25. CNN Architecture Blueprint.....	46
Pytorch implementation.....	46
Visualizing convolutional filters.....	47
26. CNN filters/feature maps look like.....	47
27. CNNs learn what features.....	47
CNNs in Pre-Deep Learning Era.....	47

28. LeNet.....	47
29. On the eve of deep learning.....	48
30. Deformable Parts Models.....	49
Modern Architectures.....	49
31. ImageNet.....	49
32. AlexNet.....	49
33. Data Augmentation.....	50
34. Generalization and Depth.....	50
35. New model to solve this problem.....	51
36. Inception block.....	51
37. Pointwise (1*1) convolution.....	51
38. Auxiliary Loss.....	51
39. VGG (Visual Geometry Group, Oxford).....	52
40. Residual Networks.....	52
41. Skip Connections (residual Networks).....	52
42. ResNets.....	52
Transfer Learning.....	53
43. Learning Visual Features.....	53
44. Transfer Learning using Embeddings.....	54
45. Fine-tuning for transfer learning.....	54
46. PyTorch implementation.....	54
Artificial Neural Networks.....	55
Neuron.....	55
1. General.....	55
Activation Function.....	55
2. Activation function.....	55
3. Linear Activation Function.....	55
4. Early Activation Functions: Perceptrons.....	55
5. Sigmoid Activation Function.....	55
6. ReLU Activation Function.....	55
Training Neural Networks.....	56
Loss Function.....	56
7. Loss function:.....	56
• computes how bad predictions are compared to the ground truth labels.....	56
8. Softmax function:.....	56
• normalizes the logits into a categorical probability distribution over all possible classes.....	56
9. Mean Squared Error (MSE):.....	57
• mostly used for regression problems.....	57
10. Cross Entropy (CE):.....	57
• mostly used for classification problems.....	57
11. Binary cross entropy (BCE).....	57
12. Forward-Pass with Error Calculations.....	57
Gradient Descent (An algorithm from optimization).....	58
13. Neural Network Layer (Vector, Matrices, Tensors).....	58
14. Neural Network Single-Layer Training.....	58
15. Delta Rule for Single Weight/Training Sample.....	58
16. Forward-pass and backward-pass.....	59
Neural Network Architectures.....	59

17. XOR.....	59
18. Backpropagation:.....	59
• Solving credit assignment problem.....	59
19. Multiple Layers with Non-Linearity.....	59
20. Neural Network Architecture.....	59
Training Artificial Neural Networks.....	60
Hyperparameters.....	60
1. General.....	60
Optimizers.....	60
2. general.....	60
3. Stochastic Gradient Descent (SGD).....	60
4. Mini-Batch Gradient Descent.....	60
5. Inefficient batch size.....	60
6. Gradient descent: N-Dimensional.....	61
7. SGD with Momentum.....	61
8. Adaptive Moment Estimation (Adam).....	61
Learning Rate.....	62
9. Learning rate:.....	62
• determines the size of the step that an optimizer takes during each iteration.....	62
Normalization.....	62
10. Reason for normalization.....	62
11. Batch Normalization.....	62
12. Layer Normalization.....	63
Regularization.....	63
13. Regularization:.....	63
• a set of techniques that you make the training task more difficult for the model.....	63
14. Dropout:.....	63
• forces a neural network to learn more robust features.....	63
15. Weight decay.....	63
16. Early Stopping with Patience.....	63
PyTorch Implementation.....	64
17. MNIST Dataset.....	64
18. ANN.....	64
19. Loss Function and Final Activation for ANN.....	65
20. PyTorch load data example.....	65
21. Forward and Backward Pass.....	65
22. PyTorch: Training and Validation Error.....	65
23. Multi-Class Classification.....	65
24. LossFunction and Softmax Activation.....	66
25. Output Probabilities.....	66
Evaluating and Debugging.....	66
26. Confusion matrix.....	66
27. MNIST 2D Visualization.....	66
28. Debugging NN.....	67
Introduction to Artificial Intelligence.....	68
1. AI.....	68
2. Machine Learning.....	68
3. Deep Learning.....	68

4. History of Deep Learning.....	68
5. Terminology Summary.....	69
6. Deep Learning applications.....	69
7. Deep Learning Caveats.....	69
8. Bias.....	69
9. Machine Learning Basis.....	69
10. Supervised Learning.....	69
11. Inductive bias (learning bias):.....	70
• the set of assumptions that used for modeling.....	70
12. Mean Squared Error (MSE):.....	70
• measures how close a regression line is to a set of data points.....	70
13. Error and loss.....	70
14. Bias versus Variance Tradeoff.....	70
15. Training and Testing Data.....	70
16. Validation and Holdout Data.....	70

Transformers

General

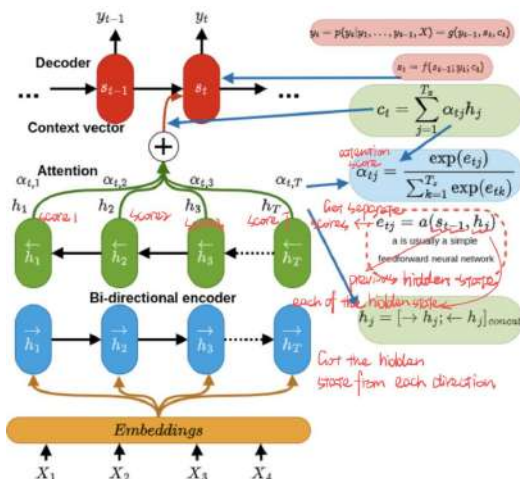
1. Quick Recap

- RNNs: model sequences -> cannot be paralleled -> inefficient
- Vanilla RNNs: cannot catch long dependencies since exploding/vanishing gradients
- LSTMs/GRUs: more preferred

Attention Mechanism

2. Attention Mechanism

- Components
 - Attention score: importance level of each 'word'
 - Aggregate the data based on this score
- Example: Classifying tweets based on attention
 - FC layer: taken in embeddings -> single score for each embedding
 - Normalize the scores: softmax
 - Weighted summation: result = sum(embedding * score)
 - Trained end-to-end with classifier
- Attention in RNNs
 - RNN without Attention: Taken the last hidden state as the representation of the whole input sentence.
 - RNN with attention: input -> embedding -> got the hidden state from each direction (Bi-directional encoder) -> attention score -> compress together -> decoder -> output



- Attention Taxonomy
 - cross-attention: capture relationship between two sequences (e.g.: translate two languages).
 - self-attention: for a given token of the input, compute attention weight for all other tokens in the sequence.
- Compute attention score

Suppose we have two embeddings $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$

We can use different methods to compute attention score between them:

- | | |
|---------------------|--|
| • Dot product | $\text{score}(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$ |
| • Cosine similarity | $\text{score}(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b} / \ \mathbf{a}\ \cdot \ \mathbf{b}\ $ |
| • Bilinear | $\text{score}(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{W} \mathbf{b}$ |
| • MLP | $\text{score}(\mathbf{a}, \mathbf{b}) = \text{Sigmoid}(\mathbf{W}[\mathbf{a}; \mathbf{b}])$ |

Transformers

3. Transformers

- Computing the similarities between raw data and the key (k), computing each key, output the value with highest similarity score



4. Attention in transformers

- Attention transformer model
 - soft retrieval: retrieves all the values \rightarrow compute their importance wrt query, based on the similarity between the query and their keys
 - values, queries, keys, are d-dimensional embeddings

$$\text{attention}(q, k, v) = \sum_i \text{similarity}(q, k_i) \times v_i$$

- Compute value, key, query (X as input)

$$Q = XW_Q, X \in \mathbb{R}^{n \times i}, W_Q \in \mathbb{R}^{i \times k}, Q \in \mathbb{R}^{n \times k}$$

$$K = XW_K, X \in \mathbb{R}^{n \times i}, W_K \in \mathbb{R}^{i \times k}, K \in \mathbb{R}^{n \times k}$$

$$V = XW_V, X \in \mathbb{R}^{n \times i}, W_V \in \mathbb{R}^{i \times v}, V \in \mathbb{R}^{n \times v}$$

- Self-attention in transformers

- New representation of each token based on weighted combination of other tokens (contextual representations)

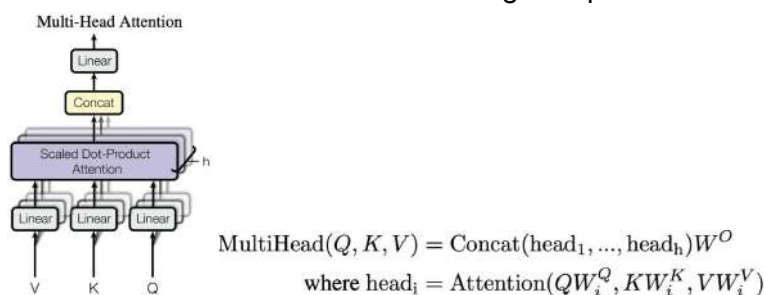
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Transpose of k
dimension of key

Same word will have different embeddings based on different context

5. Multi-head attention

- To improve the performance
 - Divide representation space to h sub-spaces
 - Run parallel linear layers and attentions
 - Concatenate them back to form the original space



6. Transformer encoders

- Each encoder layer consists of:
 - A multi-head self-attention sub-layer
 - A fully-connected sub-layer
 - A residual connection around each of the two sub-layers followed by layer normalization

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

ReLU

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

7. Positional Encoding

- When model does not have recurrent or convolutional layers \rightarrow doesn't consider the order of sequence
- Use this to allow the model to easily learn to attend by relative positions

RNN	Transformer
<ul style="list-style-type: none"> Struggling with long range dependencies Gradient vanishing and explosion Large number of training steps Recurrence prevents parallel computation 	<ul style="list-style-type: none"> Facilitate long range dependencies Less likely to have gradient vanishing and explosion problem Fewer training steps No recurrence, facilitates parallel computation

8. Pytorch implementation

```

class TweetTransformer(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetTransformer, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.encoder = TransformerEncoder(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x, pos):
        # Add GloVe vectors to positional encoding
        x = self.emb(x) + pos
        x = self.encoder(x)
        # Add embeddings from transformer encoding to get tweet embedding
        x = torch.sum(x, -1)
        # Classify
        return self.fc(x)

class TransformerEncoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(TransformerEncoder, self).__init__()
        self.linear_q = nn.Linear(input_size, hidden_size)
        self.linear_k = nn.Linear(input_size, hidden_size)
        self.linear_v = nn.Linear(input_size, hidden_size)
        self.linear_x = nn.Linear(input_size, hidden_size)
        self.attention = nn.MultiheadAttention(hidden_size, num_heads=4, batch_first=True)
        self.fc = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size))
        self.norm = nn.LayerNorm(hidden_size)

    def forward(self, x):
        q, k, v = self.linear_q(x), self.linear_k(x), self.linear_v(x)
        x = self.norm(self.linear_x(x) + self.attention(q, k, v))
        x = self.norm(x + self.fc(x))
        return x

```

Transformers for Language Modeling

9. Language Modeling

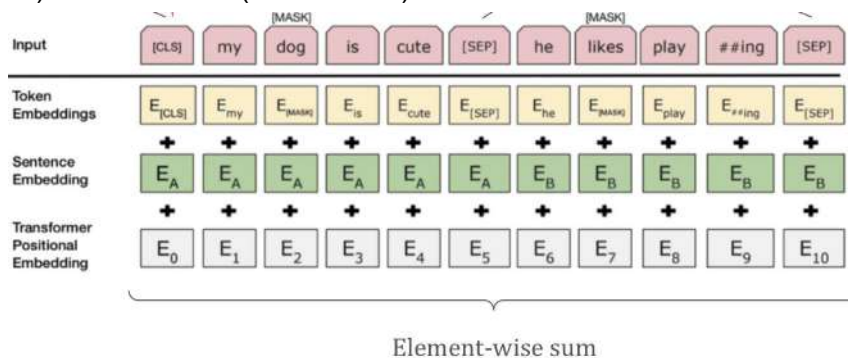
- Word2Vec/Glove
 - Learn static embeddings
 - One embedding for all senses
- Solution: use a self-supervised objective (e.g. predicting the next word) -> learn embeddings over tokens
- RNNs/Transformers
 - Learn contextual embeddings
 - 'Training' the embedding model at the same time -> embedding of a same word changes according to the sentence it appears in

10. BERT (Bidirectional Encoder Representations from Transformers)

- A Transformer model trained with two self-supervised tasks
- Shows great transfer learning capabilities
- Achieved SOTA results on various NLP tasks
- Being used in Google search engine to represent user queries and documents

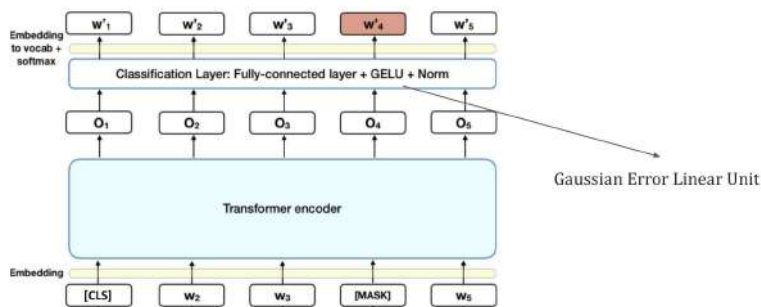
11. Input Embeddings

- [CLS] -> indicates the start of the text; specific to classification tasks
- [SEP] -> marks the end of a sentence, or the separation between two sentences
- Sentence Embedding: Specifies each token belongs to which sentence, sentence 0 (vector of 0s) or sentence 1 (vector of 1s)



12. Task 1: Masked Word Prediction

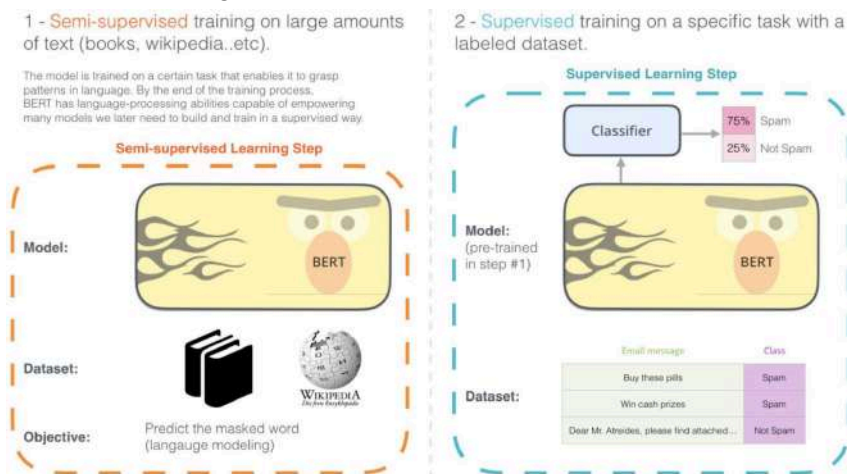
- Replace 15% of the words, at random, with [MASK] token
- Using the context of non-masked words, predict original value of [MASK] token
- Loss: computed on just the masked word (contrast with next word prediction)



13. Task 2: Next Sentence Prediction

- Determine whether two given sentences are consecutive or not in a larger corpus of text
- Create 50% positive and 50% negative pairs of sentences (less than or equal to 512 tokens)
- Loss function: BCE
- The model's final layer outputs a probability score, and this score is compared to the ground truth label

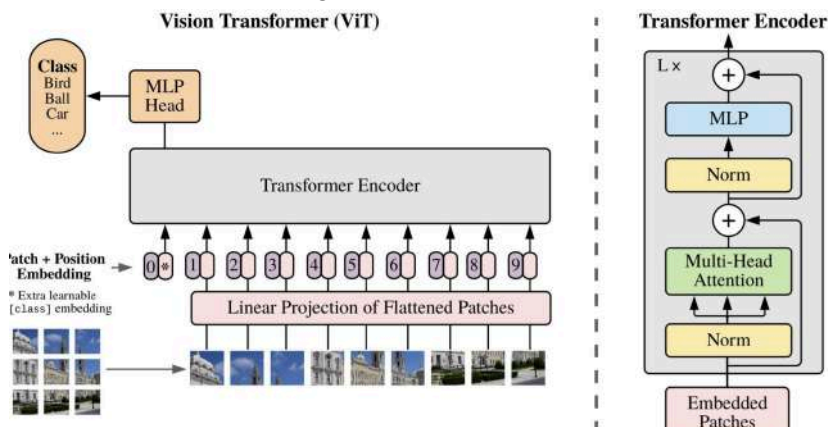
14. Transfer Learning



Transformers for Computer Vision

15. ViT (Vision Transformers)

- Achieve higher accuracies on large datasets compared to CNNs
 - Higher modeling capacity
 - Lower inductive biases
 - Global receptive fields
- CNNs better than ViTs on ImageNet in terms of model complexity or size versus accuracy
- Characteristics:
 - Train the neural network with much fewer data than transformer because the transformer is kinda learning but ViT is using
 - Taken in the image, split them into small batches (sth like slices into pieces)



Graphical Neural Networks (GNN)

1. Motivation

- Euclidean
 - CNN: 2-dimensional images
 - RNN: 1-dimensional texts
- Non-euclidean
 - GNN: e.g. molecules

Deep Sets

2. Sets

- If omit the Positional Encoding from Transformers:
 - Order-invariance / permutation-invariance: The input will be treated as a set and the learned representation won't change if you randomly shuffle the input tokens.
- Some data types that cannot be shuffled:
 - Pixels within an image
 - Words within a sentence
 - Frame within a video
 - Signals within an audio
- Model must be invariant to the order of the items <- the order can mislead the model

3. Deep Sets

- Learn embeddings for each item -> use a shared neural network to project each item to a shared space
- Learn embeddings for the set -> use an order-invariant aggregation function (e.g. sum, mean, max) to aggregate the embeddings (after aggregate is the representation of the whole dataset) into a single embedding
- Use another neural network (e.g. MLP) to project the embedding to the final space

Graphs

4. Transformers without positional encoding

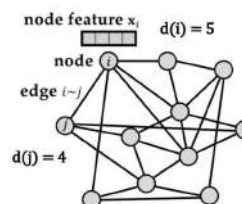
- The transformer learns an $N \times N$ attention matrix which represents a pairwise importance score -> creates a fully connected graph over the input and learns the edge weights

5. Graphs in general

- GNNs are neural networks that function on graphs
- One base:
 - Message-Passing: communicate with neighbors to update embeddings
- Graphs are order-invariant, functions on graphs must be order-invariant too
- Each graph contains an adjacency matrix, a feature matrix, and a graph
- Two characteristics:
 - Invariant -> Graph function: Output does not change in response to changes in input ordering
 - Equivariant -> Node function: Output properly changes in response to changes in input ordering

A graph $G=(V, E, X)$ is a data-structure that encodes **pair-wise interactions** or **relations** among **concepts** and **objects**:

- V is set of **nodes** representing concepts or objects
- $E \subseteq V \times V$ is a set of **edges** connecting nodes and representing relations or interactions among them
- X encodes the **node features** of each node



We can represent the edges in an **adjacency matrix A** :

Degree of a node is number of edges connecting to that node

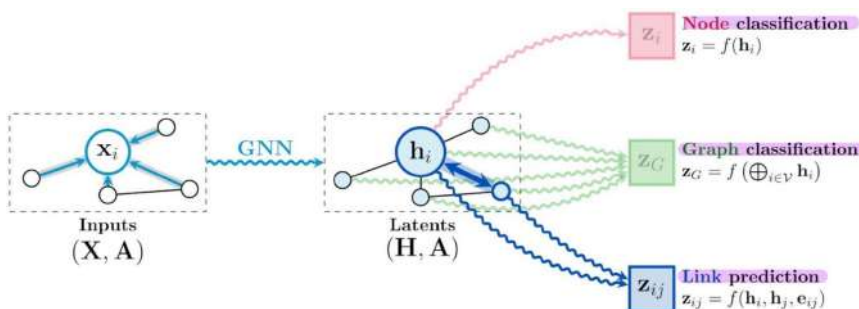
$$a_{ij} = \begin{cases} 1, & (i, j) \in E \\ 0, & (i, j) \notin E \end{cases}$$
$$d(i) = \sum_j a_{ij}$$

GNN

6. Predict links between nodes

e.g.: predicting if there should be a bond between two atoms

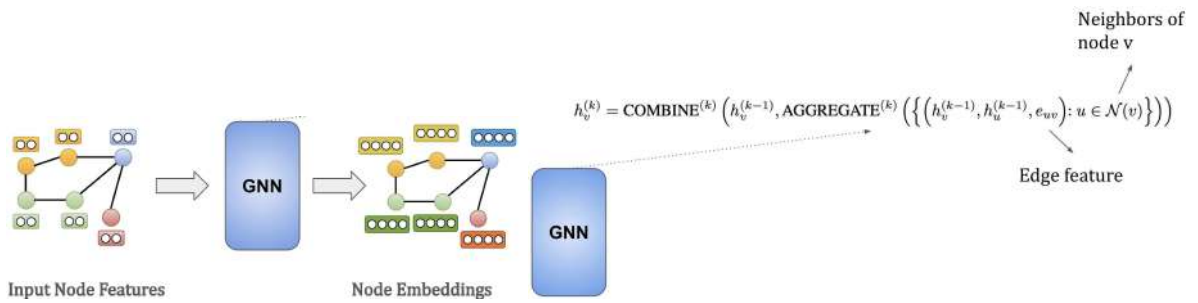
- Node classification: assign labels or categories to nodes in a graph based on their structural properties and the information from neighboring nodes.
 - Graph representation
 - Node feature: represent information about each node in the graph
 - Message passing
 - Node embeddings: represent information about each node in the graph
 - Classification layer: The output of this layer is typically passed through a softmax function to obtain probability scores for each class. Usually FC layer.
 - Training
 - Inference
- Graph classification: assign a label or category to an entire graph or network.
 - Graph representation
 - Feature extraction: graph embeddings, graph kernels, or other graph-based feature engineering methods.
 - Model selection
 - Training
 - Evaluation: Common evaluation metrics for graph classification tasks include accuracy, F1-score, or area under the receiver operating characteristic curve (AUC-ROC), depending on whether the task is binary or multiclass classification.
 - Hyperparameter tuning
 - Inference
- Link prediction: identify pairs of nodes in the network that are likely to be connected in the future or were overlooked during data collection.
 - Graph representation
 - Train-Test Split
 - Feature Engineering: For each pair of nodes without an edge in the training set (i.e., potential links), you extract or compute relevant features that describe the relationship or similarity between the nodes.
 - Model Selection
 - Training
 - Evaluation
 - Inference



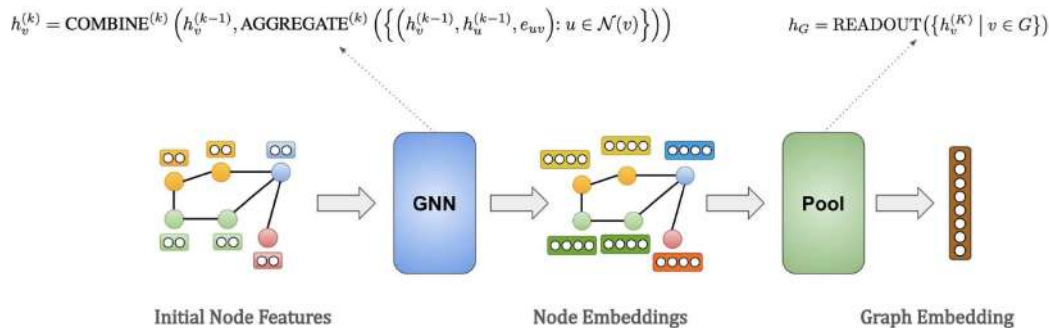
7. Message-Passing

For each node in graph:

- Aggregate embeddings of its neighbor nodes
 - Aggregate function must be an order invariant function such as sum, mean, max, attention, etc.
- Combine the aggregated embedding with the node embedding
- Update the node embedding

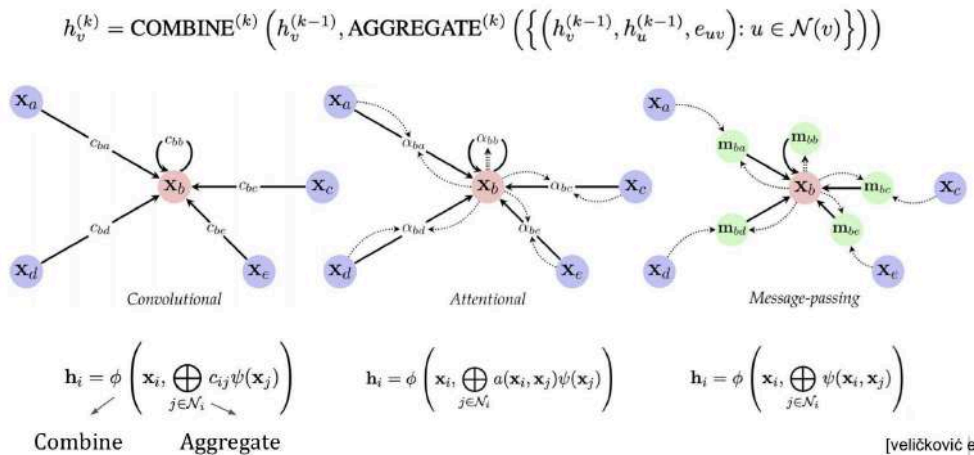


8. Read-out (graph pooling) function



9. GNN models

Different instantiations of aggregation function define different GNN models



10. Graphical convolutional networks (GCNs)

A layer of a GNN is basically a nonlinear function over node features and adjacency matrix:

$$H = f(A, X)$$

n x n *n x d*
Dimension for each node

The simplest model that we can define is:

$$H = \sigma(\underbrace{AXW}_{\text{summing up the features of their immediate neighbours}})$$

non-linearity

Weight matrix

- Limitation 1: For every node, we sum up all the feature vectors of all neighboring nodes but not the node itself.
 - Fix: Add self-loops (add the identity matrix to A)
 - $A = A + I$
- Limitation 2: A is not normalized and therefore the multiplication with A will completely change the scale of the feature vectors.
 - Fix: Symmetrically normalize A using diagonal degree matrix D such that all rows sum to one
 - $D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$

With these fixes, we define a **GCN layer** as follows:

$$H = \sigma \left(D^{-\frac{1}{2}} A D^{-\frac{1}{2}} X W \right)$$

11. Deeper with GNNs

A GCN layer updates the node embeddings based on the features of the immediate neighbors (recall multiplication with A)

We can influence the embeddings from further neighborhood by stacking GCN layers

This is analogous to increasing the receptive field in CNNs

Influencing the learned embeddings by one extra neighbourhood

$$\begin{aligned} H^{(0)} &= X \\ H^{(1)} &= \sigma \left(D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(0)} W^{(1)} \right) \\ H^{(2)} &= \sigma \left(D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(1)} W^{(2)} \right) \\ &\vdots \\ H^{(l)} &= \sigma \left(D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(l-1)} W^{(l)} \right) \end{aligned}$$

Problem: All of same importance

Node embeddings in layer 2 are computed based on contributions from 1-hop and 2-hop neighbors

12. Graph Attention Networks (GAT)

Learn an attention score between two nodes (e.g. learn the contribution weight of neighbor nodes)

1. Use a shared neural network to compute an attention score between two nodes.

$$e_{ij} = \text{NN}(h_i, h_j)$$

2. Normalize the attention scores

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

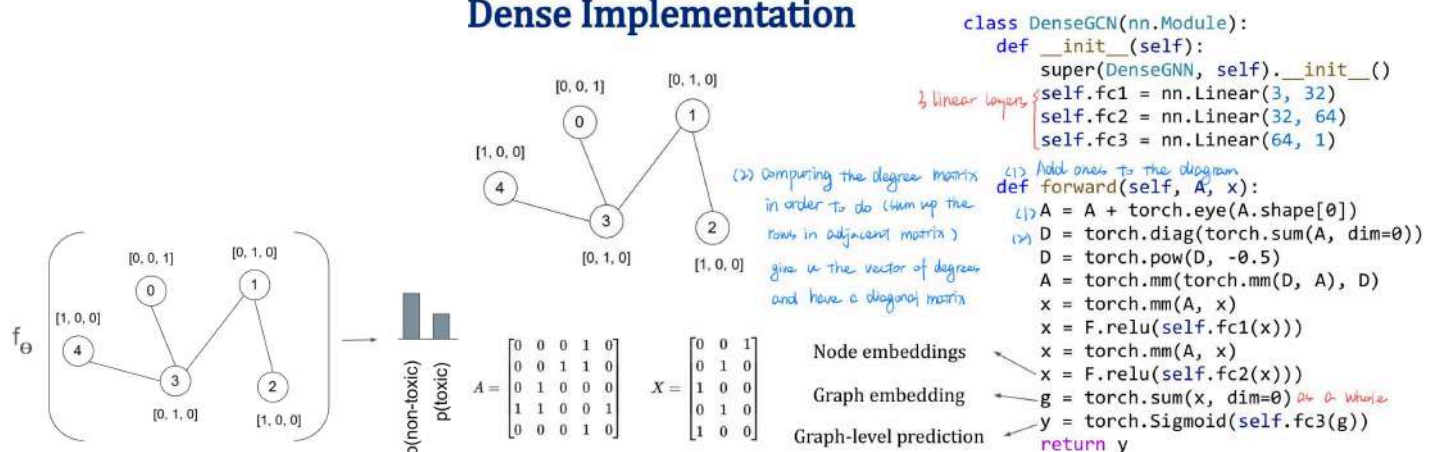
3. Update the node embeddings based on the attention score

$$h_i = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} W h_j \right)$$

Pytorch Implementation

Classify the following molecule using a GCN to a toxic/non-toxic where node features represent the atom type (Carbon, Hydrogen, Nitrogen)

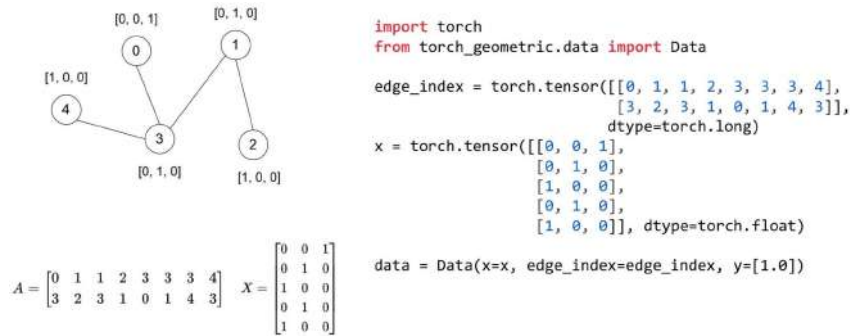
Dense Implementation



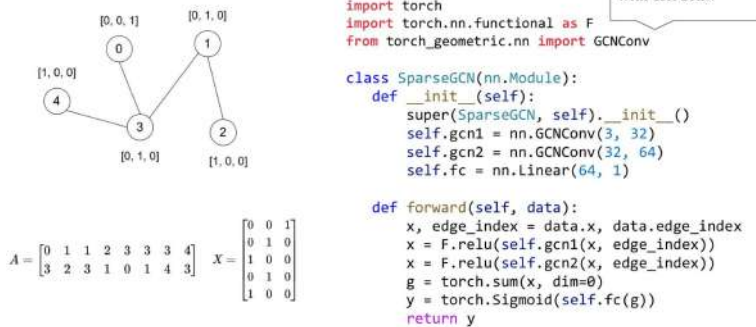
13. Dense Implementation

- Most graphs have mostly zeros in adjacent matrix
- It uses 2D space for adjacency
- We can represent this graph with only 4 edges where dense implementation represents it with 25
- Use PyTorch Geometric (PYG) to implement

14. Sparse Implementation

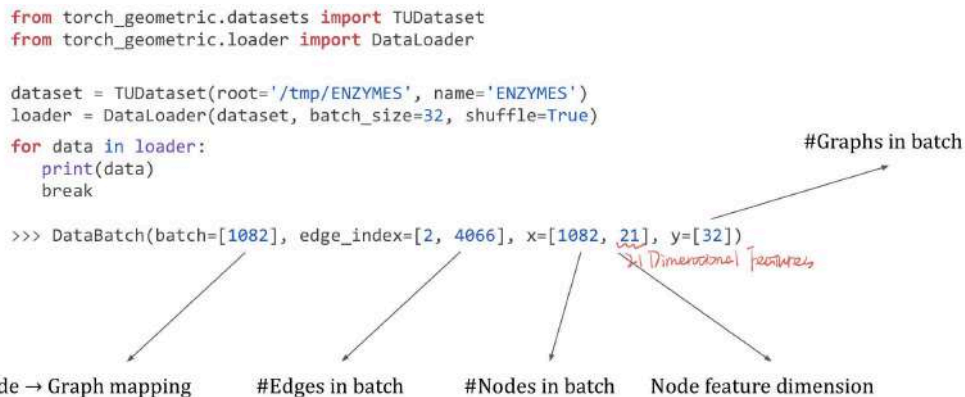


Sparse Implementation



15. DataLoader & Dataset

- Dense implementation: batching is done by creating a diagonal matrix of adjacency matrices
- Sparse implementation: uses an index vector that maps each node to its respective graph in the batch



Sparse Implementation

Now that we have a batch of graphs, we need to only sum up node embeddings corresponding to each graph

We can use PYG functions that accept the node embeddings and node-graph mapping

```
from torch_geometric.nn import GCNConv, global_add_pool

class SparseGCN(torch.nn.Module):
    def __init__(self):
        super(SparseGCN, self).__init__()
        self.gcn1 = nn.GCNConv(3, 32)
        self.gcn2 = nn.GCNConv(32, 64)
        self.fc = nn.Linear(64, 1)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.gcn1(data.x, data.edge_index))
        x = F.relu(self.gcn2(x, data.edge_index))
        g = global_add_pool(x, data.batch)
        y = torch.Sigmoid(self.fc(g))
        return y
```


Training

```
from torch_geometric.datasets import TUDataset
from torch_geometric.loader import DataLoader

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
loader = DataLoader(dataset, batch_size=32, shuffle=True)

model = SparseGCN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = nn.BCELoss()

model.train()
for epoch in range(100):
    for data in loader:
        optimizer.zero_grad()
        out = model(data)
        loss = criterion(out, data.y)
        loss.backward()
        optimizer.step()
```

Generative Adversarial Networks

Generative Models

1. Identify different models
 - Identify if a tweet is real or fake
 - Supervised task, discriminative model
 - Approximate $p(y|x)$ -> learns to approximate the conditional probability of a class given the input data
 - e.g.: classifier -> input img, output label
 - Generate a new tweet
 - Unsupervised task, generative model
 - Approximate $p(x)$ -> they are learning the probability distribution of the data itself
 - e.g.: variational autoencoder -> input encoding, output img
2. Generative learning
 - Unsupervised learning task
 - No ground truth wrt the actual task that we want to accomplish
 - Learning the structure and distribution of data, rather than labels for data
 - Loss function -> an auxiliary task that we know the answer to
3. Generative models

Used to generate new data, using some input encoding.

 - Different families of deep generative models:
 - Autoregressive Models
 - Variational AutoEncoders (VAEs)
 - Generative Adversarial Networks (GANs)
 - Flow-Based Generative Models
 - Diffusion Models
 - Unconditional Generative Models
 - Random noise as input
 - No control over what category they generate
 - Conditional Generative Models
 - User has high-level control over what the model will generate
 - Encoding
 - One-hot encoding of the target category and random noise
 - An embedding generated by another model (e.g. from CNN)
4. Problem with autoencoders
 - Vanilla autoencoders generate blurry images with blurry backgrounds <- compare pixel to pixel

Generative Adversarial Networks

5. Generative Adversarial Networks
 - The loss function of the generator is defined by the discriminator
 - Idea -> train two models together
 - Generator model
 - Fool the discriminator by generating real-looking images
 - Discriminator model
 - Distinguish between real and fake images
6. GAN model
 - Loss function for MinMax game
 - Loss function - BCE: Learn discriminator weights to maximize the probability for accurate labeling
 - Loss function - Discriminator: Learn generator weights to maximize the probability for false labeling
 - Two parts
 - Generator network

Input -> a noise vector

Output -> a generated image

- Discriminator network

Input -> an image

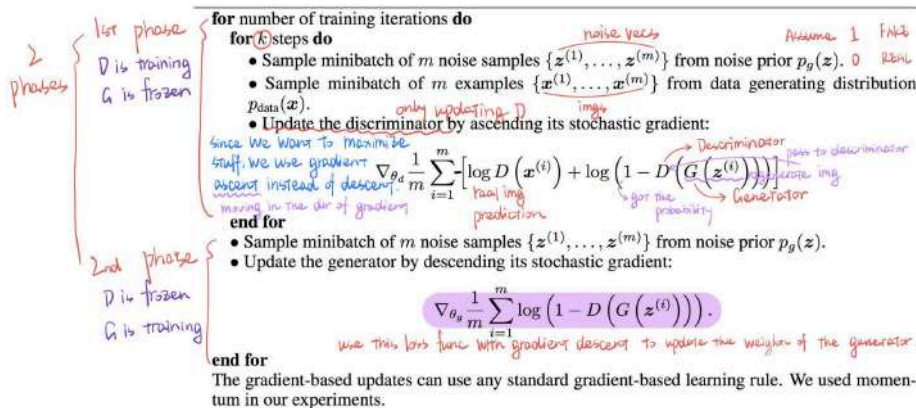
Output -> a binary label (real or fake)

- Process

- Two things evolving together, play a min-max game
- The discriminator will try to do the best job it can
- The generator is set to make the discriminator as wrong as possible

- Training

Alternate between training the D and the G



7. Pytorch implementation

- Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(28*28, 300),
            nn.LeakyReLU(0.2),
            nn.Linear(300, 100),
            nn.LeakyReLU(0.2),
            nn.Linear(100, 1))

    def forward(self, x):
        x = x.view(x.size(0), -1)
        out = self.model(x)
        return out.view(x.size(0))
```

- Generator

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 300),
            nn.LeakyReLU(0.2),
            nn.Linear(300, 28*28),
            nn.Sigmoid())

    def forward(self, x):
        out = self.model(x).view(x.size(0), 1, 28, 28)
        return out.view(x.size(0))
```

- Training the Discriminator

```
def train_discriminator(discriminator, generator, images):
    batch_size = images.size(0)
    noise = torch.randn(batch_size, 100)
    fake_images = generator(noise)
    inputs = torch.cat([images, fake_images])
    labels = torch.cat([torch.zeros(batch_size),
                        torch.ones(batch_size)])
    outputs = discriminator(inputs)
    loss = criterion(outputs, labels)
    return outputs, loss
```

- Training the Generator

```
def train_generator(discriminator, generator, batch_size):
    batch_size = images.size(0)
    noise = torch.randn(batch_size, 100)
    fake_images = generator(noise)
    outputs = discriminator(fake_images)
    # Only looks at fake outputs
    # gets rewarded if we fool the discriminator!
    labels = torch.zeros(batch_size)
    loss = criterion(outputs, labels)
    return fake_images, loss
```

Problems of Training GANs

8. Vanishing Gradients

- Discriminator as loss function for the generator,
- If the discriminator is too good, small changes in the generator weights won't change the discriminator output,
- Make no gradients, can't improve the generator

9. Mode Collapse

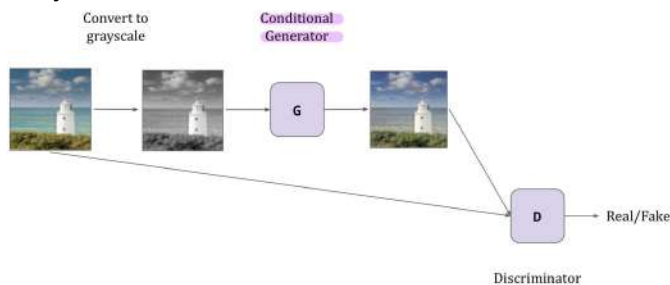
- If generator starts producing the same output (or a small set of outputs),
 - Fix: Discriminator rejects that output
- If the discriminator is trapped in local optimum, it cannot adapt to generator,
- The generator can fool it by only generating one type of data (e.g. only digit 1)

10. Failing to Converge

- Since it takes a long time to train, we use:
 - LeakyReLU Activations (training is more stable)
 - Batch Normalization
 - Regularizing discriminator weights, and adding noise to discriminator inputs

Applications of GANs

11. Grayscale to Color



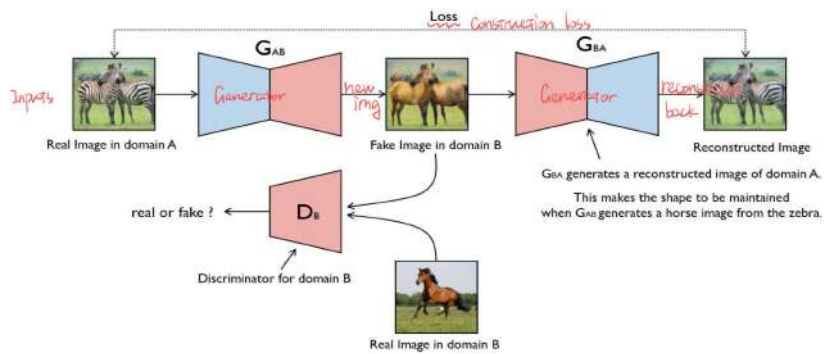
12. Conditional Generation

How could we have a GAN trained on MNIST output only specific digits?

- Data Preparation: Load the dataset, and split it into different classes
- Conditional Labeling: Each image is paired with corresponding labels
- Generator and Discriminator Architecture:
 - The generator should take random noise and the condition as inputs and generate an image corresponding to that condition.
 - The discriminator should take an image and the condition as inputs and predict whether the image matches the condition.
- Loss functions: The discriminator should take an image and the condition as inputs and predict whether the image matches the condition.
- Training
 - During training, sample random noise vectors and corresponding condition labels.
 - Input both the noise and condition to the generator to generate images.
 - Input real images with their corresponding condition labels to the discriminator.
 - Update the generator and discriminator based on their respective losses.
 - Ensure that the discriminator learns to correctly classify real images and that the generator learns to generate images conditioned on the label.
- Evaluation

13. Style Transfer

- Cycle GAN: Cycle loss is reconstruction loss between input to cyclegan and output of cyclegan to ensure consistency.



Adversarial Attacks

14. Adversarial Attacks

- Goal: Choose a small perturbation (e) on an image (x) so that a neural network (f) misclassifies ($x + e$)
- Approach: Use the same optimization process to choose e to minimize the probability $\rightarrow f(x + e) = \text{correct class}$ (e) as the parameters

15. Targeted/Non-targeted Attack

- Non-targeted attack
 - Minimize the probability, make the classifier to make mistakes $\rightarrow f(x + e) = \text{correct class}$
- Targeted attack
 - Maximize the probability, push it to make mistake at a certain type $\rightarrow f(x + e) = \text{target class}$

16. White-Box/Black-Box Attacks

- White-box attacks
 - Assumes that the model is known
 - Need to know the architecture and weights of (f) to optimize (e)
- Black-box attacks
 - Don't know the architecture and weights of (f) to optimize (e)
 - Substitute model mitigates target model with known, differentiable function
 - Adversarial attacks often transfer across models

17. Applications

- 3D Objects
- Printed Pictures
- Adversarial T-Shirts

18. Defense Against Adversarial Attack

- Failed Defenses:
 - Adding noise at test time
 - Averaging many models
 - Weight decay
 - Adding noise at training time
 - Adding adversarial noise at training time
 - Dropout

Recurrent Neural Network

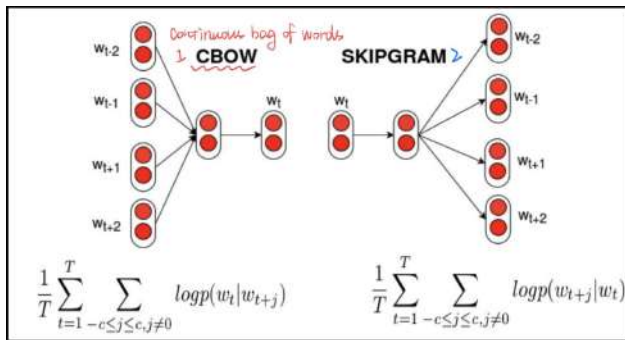
Motivation

1. Autoencoders:
 - used to learn an embedding space
 - Encoder: data -> embedding
 - Decoder: embedding -> data
2. Numerical Features
 - Superficial relationship
 - The neural network will learn it during the training and when you want to do generalization or inference or test your neural network, a big drop will see in performance, because overfitted to something that is very superficial.
 - Numbers have no relationship
 - Integer encoding is not enough when there is no order
 - An order may lead to poor performance
 - Convert words into numerical features
 - Treat each word as a unique feature
3. One-hot encoding
 - Convert word features into numerical features with one-hot encoding.
 - Turn the features into numerical features, such as if two words are completely different, the similarity value will be zero.
 - Assumes each word is completely independent
 - Problems:
 - Encoding -> Dimensionality increase
 - One-hot encoding assumes that two words are either identical / not identical, which means that it cannot capture this notion of closeness that we want to have.

Word Embeddings

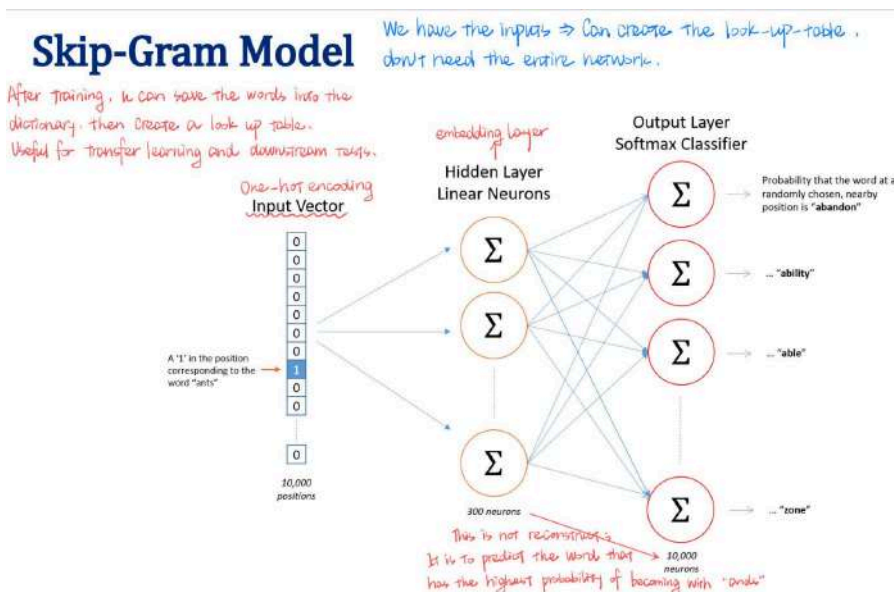
4. Characteristics
 - Words are different from images
 - Characters are not like pixels in images
 - The meaning of a word is not represented by the letters that make up the word
 - Meaning comes from context
 - Meaning comes from the sequence of characters and how they are used in conjunction with other words
 - History: coined in 2003 (Bengio et al.)
 - Two commonly used models (most famous models):
 - Word2Vec model, 2013 (Mikolov et al.)
 - GloVe vectors, 2014 (Pennington et al.)
5. Text as Sequence
 - Key idea: the meaning of a word depends on its context
 - Architecture of a word2vec model:
 - Encoder: one-hot embedding -> low-dim embedding
 - Decoder: low-dim embedding -> nearby words
 - Process: The word at the center as the word itself, the words at left and right as its context. Embedding considers both the word itself and its context in order to come up with end meanings.
6. word2vec
 - Two ways to train this model
 - Skipgram -> Predict context from target
Flip the input and output compared with CBOW. Pass the output as input (the center word), and the input as output (words to predict the center).
 - CBOW -> Predict target from context
Pass the context words to the model and the model has to predict the centre word.

- CBOW is easier because it only needs to predict one word, but the SKIPGRAM performs better



7. Skip-Gram Model

- Predict context words from target word
- Skip-Gram components need not be consecutive in the text
- Can be skipped over, or randomly selected from many documents
- Different types:
 - n-Gram: contiguous sequence of (n) items from a given text
 - k-Skip n-Gram: maximum num of skips is (k), a combination of (n) tokens or (n) words, to model real-world scenario
- Neighboring words are defined by the window size -> a hyperparameter
- Model
 - The output layer is only used for training
 - After training, only keep the weights from input to hidden layers
 - Words that have similar context words will be mapped to similar embeddings



8. CBOW (Continuous Bag of Words) Model

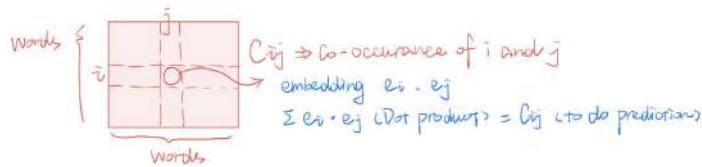
- Predict the center word from a fixed window size of context words
 - Pass in the context words, predict the center word

9. CBOW versus Skip-Gram

- Skip-Gram
 - Works well with small datasets
 - Better semantic relationships (cat & dog)
 - Better representation of less frequent words
- CBOW
 - Trains faster than Skip-Gram as the task is simpler
 - Better syntactic relationships (cat & cats)
 - Better representation of more frequent words

10. GloVe: Utilizes the global statistics

- Compute co-occurrence frequency counts for each word
 - Represented as a matrix where element $X(i,j)$ demotes the number of times word (i) appears in the context of word (j)
- Optimization: Inner product of word vectors should be a good predictor of co-occurrence frequency



11. PyTorch GloVe Embeddings

Use **torchtext** package to load pre-trained GloVe embeddings

First time you run it will load an 862MB file containing pretrained embeddings

6B was trained on Wikipedia 2014 corpus

```
import torch
import torchtext

glove = torchtext.vocab.GloVe(name='6B', dim=50)
glove['cat']

tensor([0.4769, -0.0846, ...])
```

Distance Measures

12. Measure of distance in the embedding space

Euclidean Distance → L2-norm of embeddings

$$D(\vec{X}, \vec{Y}) = \|\vec{X} - \vec{Y}\| = \sqrt{\sum_{i=0}^d (x_i - y_i)^2}$$

Cosine Similarity → cosine of the angle between embeddings (invariant to magnitude)

$$\text{Sim}(\vec{X}, \vec{Y}) = \cos(\theta) = \frac{\vec{X} \cdot \vec{Y}}{\|\vec{X}\| \|\vec{Y}\|} = \frac{\sum_{i=0}^d x_i y_i}{\sqrt{\sum_{i=0}^d x_i^2} \sqrt{\sum_{i=0}^d y_i^2}}$$

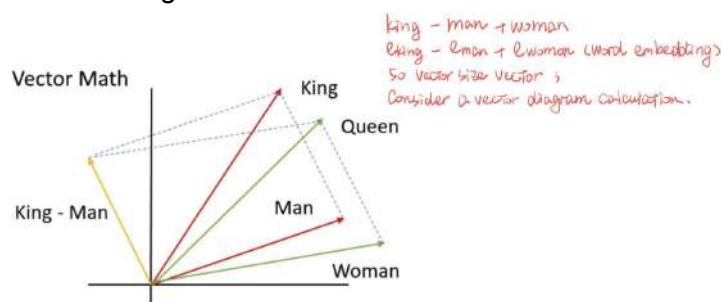
Euclidean Distance:

```
torch.norm(glove['cat']-glove['dog'])
```

Cosine Similarity:

```
torch.cosine_similarity(
    glove['cat'].unsqueeze(0), glove['dog'].unsqueeze(0))
```

13. Word Analogies



14. Bias in Word Embeddings

- Machine learning models are biased
- ML models learn the biases present in the data it is trained on

Language Models

15. Language Modelling

- Explain
 - Have a model that can learn the probability distribution over a given language or other stuff

- Because the language model needs to understand the semantic to be able to predict the probability
- Learning probability distribution over sequences of words
 - Text understanding
 - Text generation

16. Working with Text

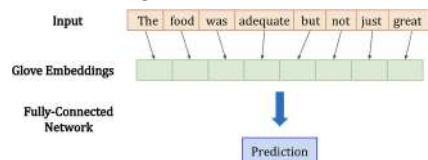
- Difficult for text:
 - The meaning doesn't depend on characters
 - It has dependencies

17. Sentiment Analysis

- Goal: Given a piece of text, identify the sentiment that the text conveys
- Dataset: Sentiment140
 - Split the tweet into words by white-space
 - Look up the GloVe embedding for each word, ignoring words that don't have embeddings
 - Add up the word embeddings to obtain an embedding for the entire tweet
 - The tweet embedding will be the input to a fully connected neural network

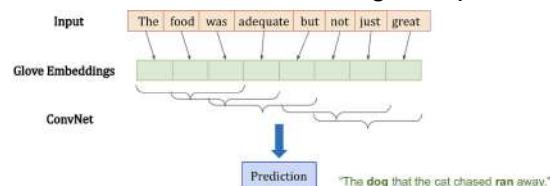
18. Limitations

- The model does not take into account the order of words
- Fix 1:
 - Concatenate the word embeddings -> train a neural network that takes the concatenated embedding as input



- Drawbacks:
 - ~Fixed-Length input: If your input text varies significantly in length, you may need to pad or truncate the sequences to a specific length.
 - ~Loss of sequential information
 - ~Increased Dimensionality:
 - If you have a large vocabulary and embeddings with high dimensions, the concatenated representation can become very high-dimensional.
 - lead to increased computational complexity and the risk of overfitting, especially when you have limited training data.
 - ~Out of vocabulary words (OOV)
 - ~Semantic Gap: Concatenating word embeddings treats all words equally, regardless of their importance in the task

- Fix 2:
 - Concatenate the word embeddings -> train a 1D convolutional neural network that takes the concatenated embedding as input



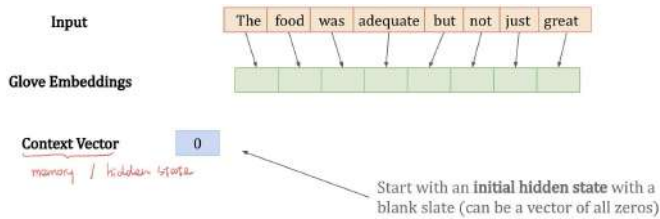
- Drawbacks:
 - ~Lack of contextual information
 - ~Local context only: 1D CNNs are effective at capturing local patterns in the input, but they may struggle with capturing long-range dependencies or global context in the text.

~Generalization: Depending on the choice of hyperparameters and architectural details, 1D CNNs may not generalize well to diverse text data or handle out-of-distribution examples effectively.

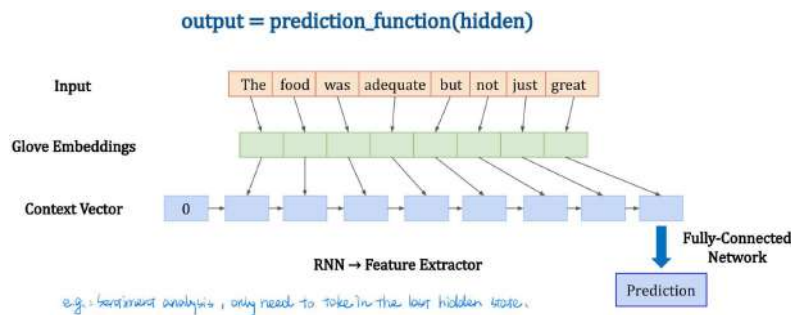
Recurrent Neural Networks (RNNs)

19. RNNs

- Take in variable-sized sequential input
- Remember things over time, or have some sort of memory or state

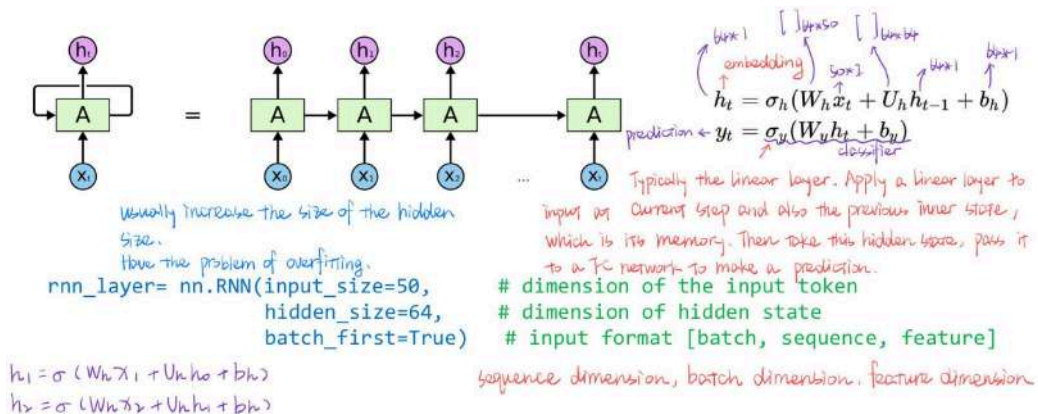


- Updating Hidden State: updated based on the previous hidden state and the input using the same neural network as before (weight sharing)
- Last Hidden State: Continue updating the hidden state until we run out of tokens
 - Use the last hidden state as input to a prediction network
 - The whole thing is end-to-end, when you train, you get gradients here and you backpropagate through the neural network
 - The last hidden state (memory) has compressed all the information that you have seen so far



20. RNN Layers

- In each step, it receives input and its previous hidden state and updates it, then the next input and another update make the overall update. -> This is a feedback loop, there is a recurrence.



21. PyTorch Implementation

- RNN Architecture

```

class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, _ = self.rnn(x, h0)
        # Pass the output of the last step to the classifier
        return self.fc(out[:, -1, :])

model = TweetRNN(50, 64, 2)

```

• RNN Training

```

def train(model, train, val, n_epochs=5, lr=1e-5):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    for epoch in range(n_epochs):
        for tweets, labels in train:
            optimizer.zero_grad()
            pred = model(tweets)
            loss = criterion(pred, labels)
            loss.backward()
            loss.step()

```

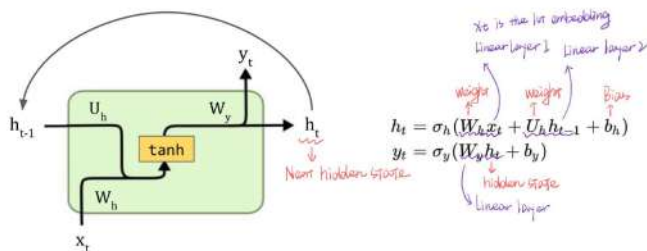
22. Sequential Learning

- In image, we do not want to learn different weights for every pixel
 - CNNs use convolutional filters with parameter sharing
 - CNN reuses convolutional filters for every pixel
- In sequence, we do not want to learn different weights for every token
 - RNNs use a shared neural network to update hidden state
 - Reuse the RNN module for every token in the sequence
 - Keep the context of the previous tokens encoded in the hidden state (h)

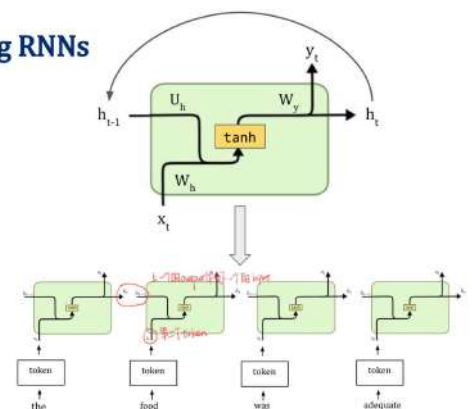
23. Different RNN models

RNN

If we consider the **concatenated input/hidden** and **output/hidden** vectors as simply input/output, forward path in RNN is simply a **fully-connected NN**

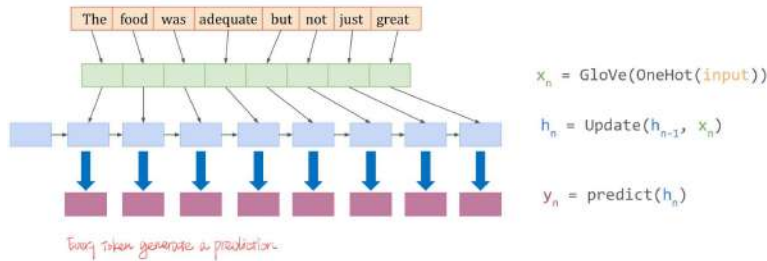


Unrolling RNNs

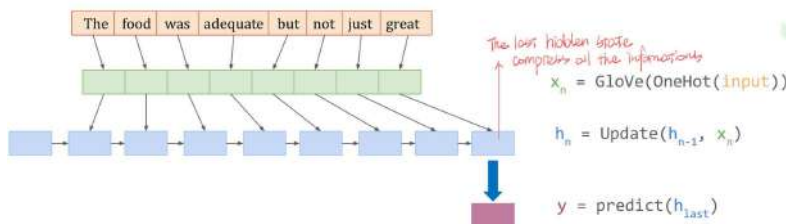


24. Different types of prediction

Token-Level Predictions



Sequence-Level Predictions



Limitations of Vanilla RNNs

25. Problem

- If RNNs unrolled onto a long sequence:
 - RNNs can be very deep \rightarrow Depth = Length of sequence
- 2 related problems with vanilla RNNs
 - Not good at modeling long-term dependencies
 - Hard to train due to vanishing/exploding gradients

26. Exploding/vanishing gradients

- Problem explanation

Suppose update function is a simple linear model. For simplicity, let's ignore inputs:

$$h_0 \xrightarrow{W_h} h_1 \xrightarrow{W_h} \dots \xrightarrow{W_h} h_t \quad h_t = W_h h_{t-1}$$

We can write this for all time-steps as:

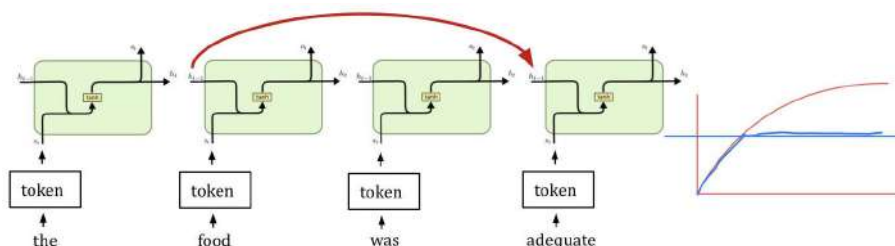
$$h_t = (W_h)^t h_0$$

Then we have:

- Exploding gradients $h_t \rightarrow \infty$ if $|W_h| > 1$ τ is very big, $|W_h|$ slightly larger than 1.
- Vanishing gradient $h_t \rightarrow 0$ if $|W_h| < 1$

- Fix

- Gradient clipping \rightarrow exploding gradient: if gradient is greater than a threshold, set the gradient to threshold
- Skip-connection \rightarrow vanishing gradient:
 - Skip connections to all previous states \rightarrow too expensive \rightarrow preserve the hidden state/context over the long term



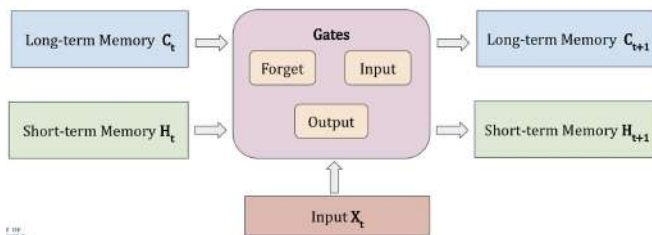
LSTMs & GRUs

27. Gating Mechanism

- Approximate skip-connections to all previous states
 - Learn to weight previous states differently instead (soft skip-connections)
- Use gates
 - Learn to update the context selectively
- Gating mechanism controls how much information flows through
- Suppose X is a vector, control how much of X to pass to next step by:
 - Sigmoid or Tanh
 - A neural network

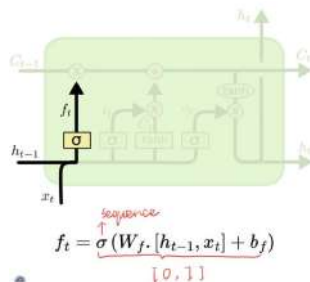
28. Long Short-Term Memory (LSTM)

- Components
 - Long-term memory (cell state)
 - Short-term memory (context or hidden state): Assigning different weights to different hidden states
- Use three gates to update the memories

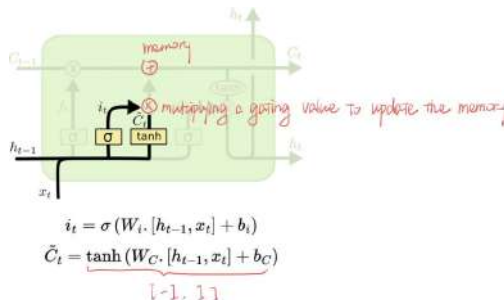


29. Different gates for LSTM

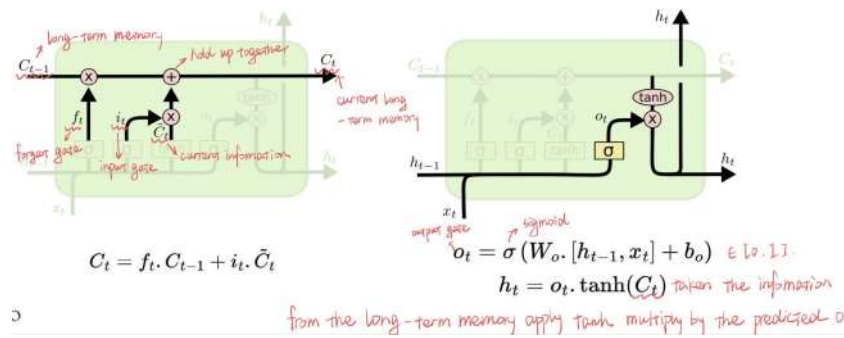
- Forget gate (long-term memory)
 - How many of the historical memory should I forget



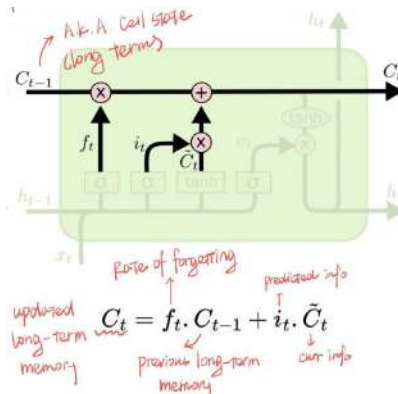
- Input Gate (long-term memory)
 - How much the current input should contribute to the memory



- Output gate (short-term memory)
 - In order to update the short-term memory
 - How much of the updated long-term memory should construct the short-term memory

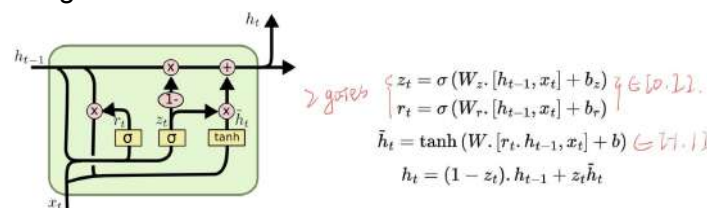


- Updated long-term memory: the amount of past that is remembered (decided by forget gate) combine with the memory that was just created (decided by input gate)



30. Gated Recurrent Unit (GRU)

- Cheaper than LSTM since one less hidden state and one less gate
- Characteristics:
 - Combine forget and input gates into an update gate
 - Merges cell state and hidden state



31. LSTM/GRU versus RNN

- LSTMs/GRUs
 - can be trained on longer sequences
 - Are much better at learning long-term relationships
 - Easier to train
 - Achieve better performance than vanilla RNNs
- In long-term, RNNs will stick to the accuracy, but LSTM/GRU will improve even more

32. PyTorch implementation

- RNN

```
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, _ = self.rnn(x, h0)
        # Pass the output of the last step to the classifier
        return self.fc(out[:, -1, :])
```

- GRU


```
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.GRU(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, _ = self.rnn(x, h0)
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])
```

- LSTM

```
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

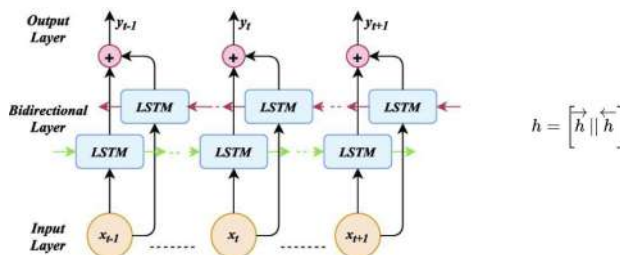
    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, _ = self.rnn(x, (h0, c0))
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])
```

*Initial cells states
Long-term memory*

Deep & Bidirectional RNNs

33. Bidirectional RNNs

- A typical state in an RNN (RNN, GRU, LSTM) relies on the past and the present
- One from left to right, the other from right to left, only take the sum of all the hidden state



- When a prediction depends on the past, present, and future, we can exploit the future to improve performance
 - e.g.: machine translation

34. Deep RNNs

- Stack RNN layers to learn more abstract representations
 - First layers: better for syntactic tasks
 - Last layers: better on semantic tasks

35. PyTorch implementations

PyTorch Details

the dimensions of the

```
self.rnn = nn.GRU(input_size=64, embedding
                  hidden_size=256, encoding
                  batch_first=True, sequence, batch, hidden
                  num_layers=4,
                  bidirectional=True)
By default, bidirectional=False, num_layers=1, batch_first=False

h0 = torch.zeros(2x4, x.size(0), self.hidden_size)

output, h_n = self.rnn(x, h0)
```

Batch size
sequence length

tensor of shape (N, L, D_{out} × H_{out}) containing the output features (h_t) from the last layer of the GRU, for each t
2x4
bidirectional: D × 2

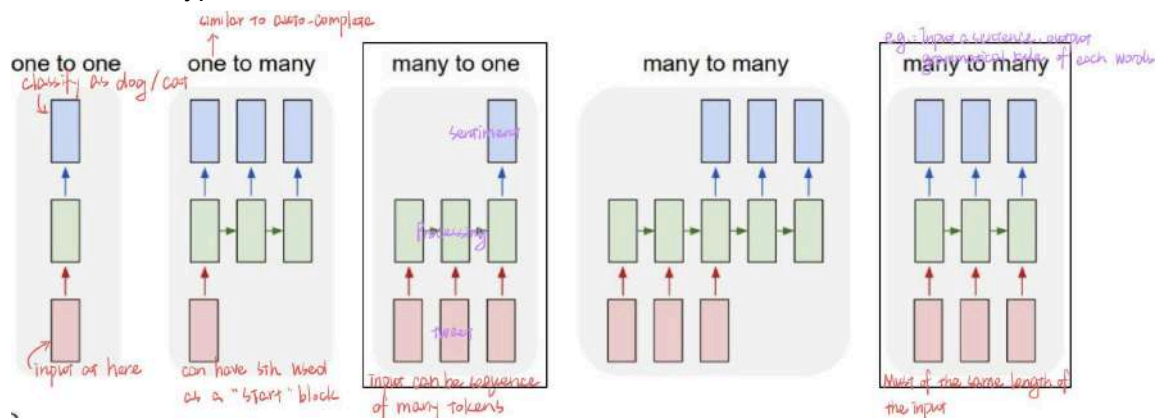
tensor of shape (D_{num_layers}, N, H_{out}) containing the final hidden state for the input sequence.

N = batch size
 L = sequence length
 $D = 2$ if $\text{bidirectional} = \text{True}$ otherwise 1
 H_{in} = input_size
 H_{out} = hidden_size

Output
h_n

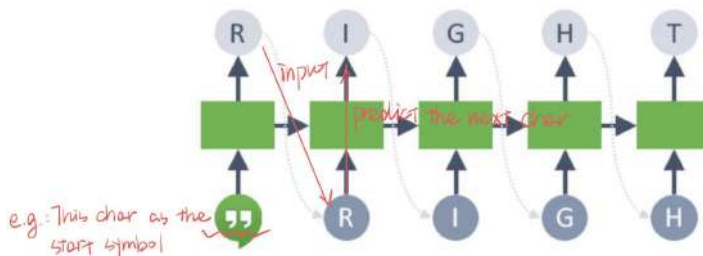
Sequence-to-Sequence Models

36. RNN Model Types



37. Hidden State Differences

- RNNs for prediction (Encoder): compressing all information about the past
 - Process tokens one at a time
 - Hidden state represents all the tokens read this far
- RNNs for generating sequences (decoder): Hidden state in each time step compressing all the info about the future
 - Generate tokens one at a time
 - Hidden state is a representation of all the tokens to be generated
- Autoregressive: Say one word then say the next word based on the previous word



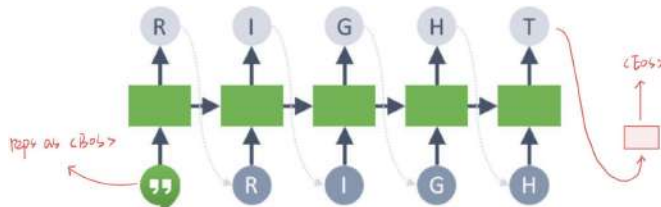
38. Sequence-to-Sequence RNNs

- Summarize:
 - Use this promo code (<BOS>/<EOS>) to communicate with the neural network to notify starting and ending,
 - Then use cross-entropy to compare the loss and do the backpropagation,
 - Then introduce teacher forcing to make sure that each step we receive the perfect ground truth predictions rather than pollution that were done by the neural network.

- With variational autoencoders, we have this randomness -> pass a random number into the decoder to generate an image for us, want to have diversity
- Different types:
 - Teacher-forcing: Training-time behaviour must be changed
 - Sampling and temperature scaling: Inference-time behavior also changes

39. During training

- When to stop/finish a generated sequence
 - <BOS>: indicate the beginning of sequence
 - <EOS>: indicates the end of sequence
- In each step, RNN receives an input which is the previous prediction and is predicting a class, so evaluate at cross entropy loss at each generation step. Each character generates a loss. Average of the losses will be the overall loss of the sequence.



40. Teacher forcing

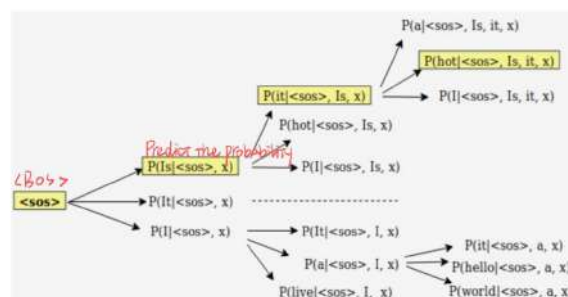
- Reason
 - Basically, we are increasing the noise and accumulating the error so at the far end it is difficult to predict the last character because the noise.
- Process
 - In each step during the training make the prediction and compare with the ground truth label, and compute the cross entropy loss but don't feed the prediction as the next input.
 - Basically, instead of passing the previous prediction, pass the ground-truth label to there as the input to the next step.

41. During Inference

- Problem: Always selecting the token with the highest probability won't work well
 - In practice, this greedy approach results in lots of grammatical errors
 - Using a generative model: We want diversity not deterministic behavior
- Fix: sample from the predicted distributions
 - Greedy Search: selects the token with highest probability as the generated token

$$\max p(t_1, t_2, \dots, t_n) = \max p(t_1) \times p(t_2) \times \dots \times p(t_n)$$
 - Beam Search: looks for a sequence of tokens with the highest probability within a window

$$\max p(t_1, t_2, \dots, t_n) = \max p(t_1) \times p(t_2|t_1) \times \dots \times p(t_n|t_{n-1}, \dots, t_2, t_1)$$

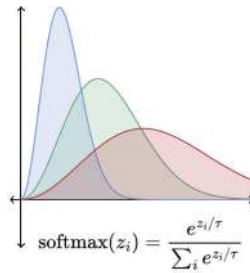


- Softmax Temperature Scaling: helps with the problem of over-confidence in neural networks by scaling the input logits to the softmax with a temperature.
 ~Low Temperature (larger logits, more confident):
 Higher quality samples, less variety;
 If set the temperature to low value, your distribution will be moved toward a one-hot representation.

~High Temperature (smaller logits, less confident):

Lower quality samples, more variety;

If set to high temperature, the distribution will be very similar to uniform distribution



42. PyTorch implementation

• Text Generator

```
class TextGenerator(nn.Module):
    def __init__(self, vocab_size, hidden_size, n_layers=1):
        super(TextGenerator, self).__init__()
        # Identity matrix for generating 1-hot vectors
        self.identity = torch.eye(vocab_size)
        # Recurrent neural network
        self.rnn = nn.GRU(vocab_size, hidden_size, batch_first=True)
        # A FC layer outputting a distribution over the next token
        self.decoder = nn.Linear(hidden_size, vocab_size)

    def forward(self, inp, hidden=None):
        # Generate 1-hot vectors of input
        inp = self.identity[inp]
        # Get the next output and hidden state
        output, hidden = self.rnn(inp, hidden)
        # Predict distribution over next tokens
        output = self.decoder(output)
        return output, hidden
```

• Training Text Generator

```
def train(model, data, batch_size=1, num_epochs=1, lr=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    data_iter = torchtext.data.BucketIterator(data,
        batch_size=batch_size,
        sort_key=lambda x: len(x.text),
        sort_within_batch=True)

    for _ in range(num_epochs):
        Avg_loss = 0
        for (tweet, lengths), label in data_iter:
            target = tweet[:, 1:] for ground truth label
            inp = tweet[:, :-1]
            optimizer.zero_grad()
            output, _ = model(inp)
            loss = criterion(output.reshape(-1, vocab_size), target.reshape(-1))
            loss.backward()
            optimizer.step()
```



• Sampling Text Generator

```
def sample(model, max_len=100, temperature=0.8):
    generated_sequence = ''
    inp = torch.Tensor([vocab_stoi['<BOS>']]).long()
    hidden = None
    for p in range(max_len):
        output, hidden = model(inp.unsqueeze(0), hidden)
        # Sample from the model as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = int(torch.multinomial(output_dist, 1)[0])
        # Add predicted character to string and use as next input
        predicted_char = vocab_itos[top_i]
        if predicted_char == '<EOS>':
            break
        generated_sequence += predicted_char
        inp = torch.Tensor([top_i]).long()
    return generated_sequence
```

Unsupervised Learning

Motivation

1. Challenges with Supervised Learning

- Requires large amounts of labeled data
- Obtaining labeled data is expensive
 - Medical tests are expensive -> require a specialist to review them
 - Chemical data collection -> wet-lab tests are time-consuming
- For more scenarios, there is a lot more unlabeled data than labeled

2. Feature Clustering:

- Learn the underlying patterns, then need a few examples just to label.

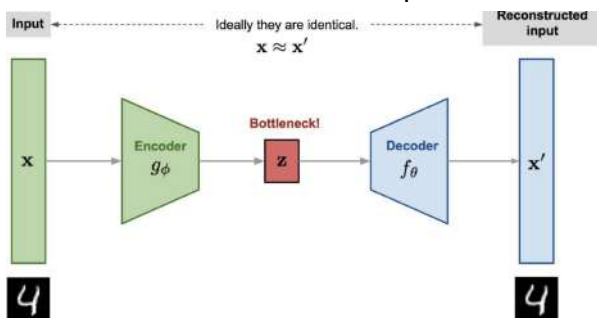
3. Definitions

- Unsupervised Learning
 - Learning patterns from data without human annotations
 - e.g.: clustering, density estimation, dimensionality reduction
- Self-supervised Learning
 - Use the success of supervised learning without relying on human-provided supervision (automatic supervision)
 - e.g.: mask part of the input and predict the masked information
- Semi-supervised Learning
 - Learning from data that mostly consists of unlabeled samples
 - A small amount of human-labeled data is available as well

Autoencoders

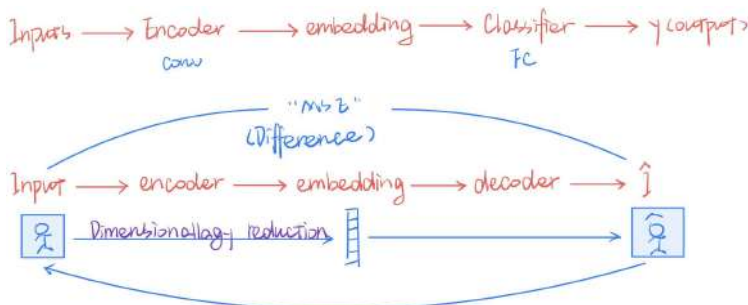
4. General information

- The number of outputs is the same as the inputs
- Hourglass shape creates a bottleneck layer, lower dimensional representation
- It is forced to learn the most important features in the input data and drop the unimportant ones



Find efficient representations of input data that could be used to reconstruct the original input:

- Encoder:
 - Converts the inputs to an internal representation
 - Dimensionality reduction
- Decoder:
 - Converts the internal representations to the outputs
 - Generative network
- Given a picture, encode to vector, embedding, then decode to get the original picture, compare with the input image to see the differences.



5. Applications

- Feature Extraction
- Unsupervised Pre-training
- Dimensionality Reduction
- Generate new data
- Anomaly detection -> Autoencoders are bad at reconstructing outliers

6. PyTorch implementations

- Error rate: the number of things that you misclassified divided by the whole number of training data.

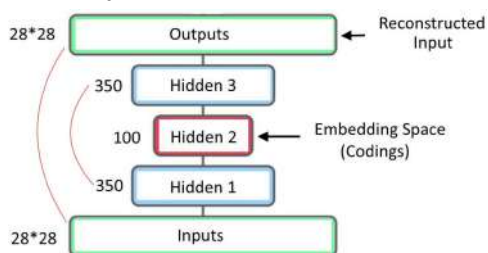
```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        encoding_dim = 32
        self.encoder = nn.Linear(28 * 28, encoding_dim)
        self.decoder = nn.Linear(encoding_dim, 28 * 28)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        x = self.encoder(flattened)
        # sigmoid for scaling output from 0 to 1
        x = F.sigmoid(self.decoder(x))
        return x

criterion = nn.MSELoss()
```

7. Stacked Autoencoders

- Usually want symmetric structure between the encoder and decoder.
- Autoencoders can have multiple hidden layers: stacked (deep) autoencoders
- Output and Input size need to be the same
- One way to ensure that an autoencoder is properly trained is visualizing reconstructions



8. Denoising Autoencoders

- Noise can be added to the input images of the autoencoder to force it to learn useful features
- Autoencoder is trained to recover the original, noise-free inputs
- Prevents it from trivially copying its inputs to its outputs, has to find patterns in the data

PyTorch Implementation

```
# how much noise to add to images
nf = 0.4 introduce noise factor

# add random noise to the input images
noisy_img = img + nf * torch.randn(*img.shape)

# Clip the images to be between 0 and 1
noisy_img = np.clip(noisy_img, 0., 1.)

# compute predicted outputs using noisy_img
outputs = model(noisy_img)

# the target is the original img
loss = criterion(outputs, img)
```

9. Generating New Images with Interpolation

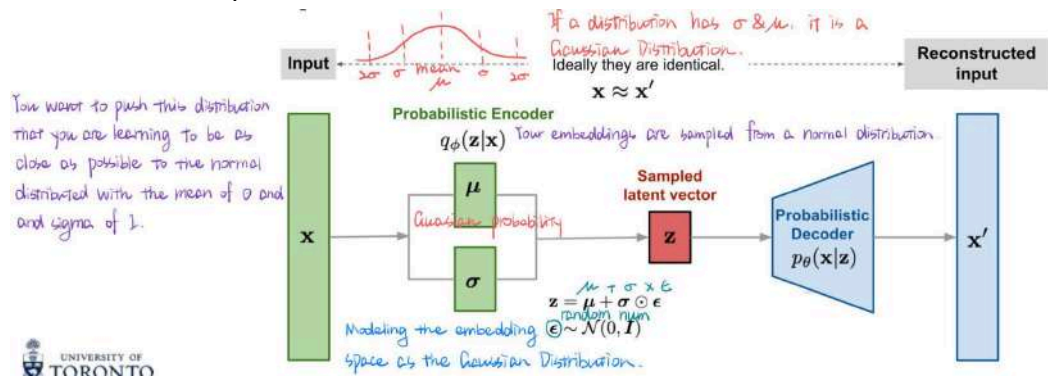
- First compute low-dimensional embeddings of two images
- Then interpolate between the two embeddings and decode those
- Interpolated coding results in new images that are somewhere in between the two starting images
- Latent space: The space where the embedding lives on

- If we randomly select a coding:
 - The latent space in autoencoders can become disjoint and non-continuous

Variational AutoEncoders (VAE)

10. Characteristics

- Encoder generates a normal distribution with mean μ and a standard deviation σ instead of a fixed embedding.
 - An embedding is sampled from the distribution and decoder decodes the sample to reconstruct the input



- Before, the models we learned were all deterministic. If you give convolutional network train, if you pass the same image twice, the output will be identical.
- Distribution calculation

We want the encoder distribution $q_\phi(z|x) = \mathcal{N}(\mu, \sigma)$ to be close to prior $p(z) = \mathcal{N}(0, I)$

We can use **Kullback-Leibler (KL)** divergence to measure the difference between two distributions $P(X)$ and $Q(X)$:

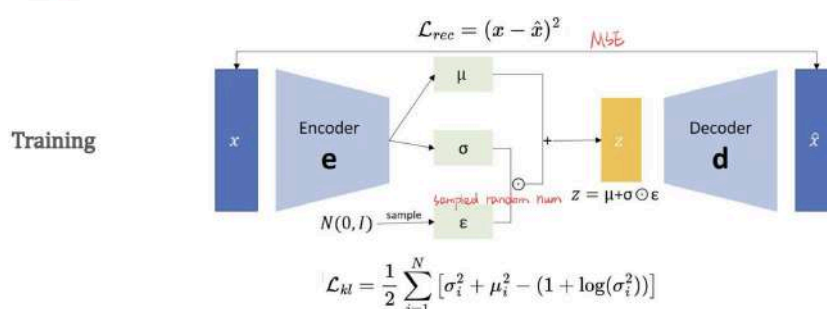
$$D_{KL}(P||Q) = \sum_{x \in X} p(x) \log \left(\frac{p(x)}{q(x)} \right)$$

cross entropy

If we plug-in the **encoder distribution** and the prior into KL-divergence of two **multivariate Gaussians**, we get:

$$D_{KL}(p||q) = \frac{1}{2} \sum_{i=1}^N [\mu_i^2 + \sigma_i^2 - (1 + \log(\sigma_i^2))]$$

What's the difference between the distribution that the encoder is learning and the normal distribution with $\mu = 0$.



- Probabilistic
 - For the same input, it will give you different results every time
 - Their outputs are partly determined by chance even after training
- Generative
 - It can generate an infinite number of examples for you, that were not part of the training data

- They can generate new instances that look like they were sampled from the training set

11. Different types autoencoders

- Regular Autoencoders -> have problem with overfitting
- Noisy autoencoders -> have problems if generate stuff for the smoothness of the embedding
- Variational autoencoders

Convolutional autoencoder

12. Convolutional autoencoder

- Use spatial information
 - Encoder: Learns visual embedding using convolutional layers
 - Decoder: Up-samples the learned visual embedding to match the original size of the image
- ~Up-sampling: It involves increasing the size of the data from a lower resolution to match a higher resolution. In the context of image processing, this often means making an image or a feature map larger.

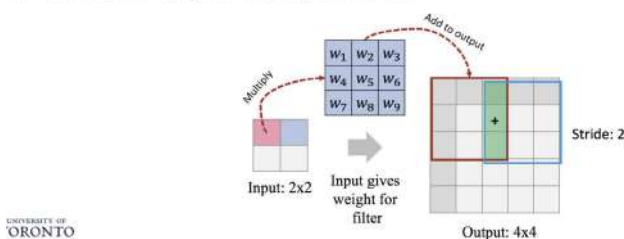
13. Transposed Convolution

- Similar to convolution, but the inverse effect of a convolution
- Instead of mapping $k \times k$ pixels to 1, they can map from 1 pixel to $k \times k$ pixels

$$o = (i - 1) \times s + (k - 1) - 2p + op + 1$$

output dimension input dimension stride kernel size padding output padding

1. Take each pixel of your input image
2. Multiply each value of your kernel with the input pixel to get a weighted kernel
3. Insert it in the output to create an image
4. Where the outputs overlap sum them



14. Padding

- Output padding
 - Output padding is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side
 - It is only used to find output shape but does not actually add zero padding to output
 - When stride > 1. Conv2d maps multiple input shapes to the same output shape
- The effect is the opposite of what happens with the convolution layers
 - Compute the output as normal
 - Remove rows and columns around the perimeter

15. Strides

- The effect is opposite from what happens with the convolution layers
- Increasing the stride results in an increase in the upsampling effect

16. PyTorch implementations

- Compare

```
conv = nn.Conv2d(in_channels=8, out_channels=8, kernel_size=5)
convt = nn.ConvTranspose2d(in_channels=8, out_channels=8, kernel_size=5)

x = torch.randn(2, 8, 64, 64)
y = conv(x)
y.shape

x = torch.randn(2, 8, 64, 64)
y = convt(x)
y.shape # should be same as x.shape
```

Diagram: A red arrow labeled "input" points from `x` to `convt(y).shape`. A red arrow labeled "output" points from `y.shape` to `torch.Size([2, 8, 60, 60])`. The output shape `torch.Size([2, 8, 60, 60])` is shown below the first code block.

- Transpose padding

```
convt = nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=5, padding=2)

x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

`torch.Size([32, 8, 64, 64])`

$$o = (i - 1) \times s + (k - 1) - 2p + op + 1$$

- Add a stride to the convolution to increase our resolution

```
convt = nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=5, stride=2, padding=2)
```

```
x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

`torch.Size([32, 8, 127, 127])`

$$o = (i - 1) \times s + (k - 1) - 2p + op + 1$$

- Output padding type adds an additional row and column to the output

```
convt = nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=5, stride=2, padding=2, output_padding=1)
```

```
x = torch.randn(32, 16, 64, 64)
y = convt(x)
y.shape
```

`torch.Size([32, 8, 128, 128])`

$$o = (i - 1) \times s + (k - 1) - 2p + op + 1$$

- Others

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )
```

```

def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x

def embed(self, x)
    return self.encoder(x)

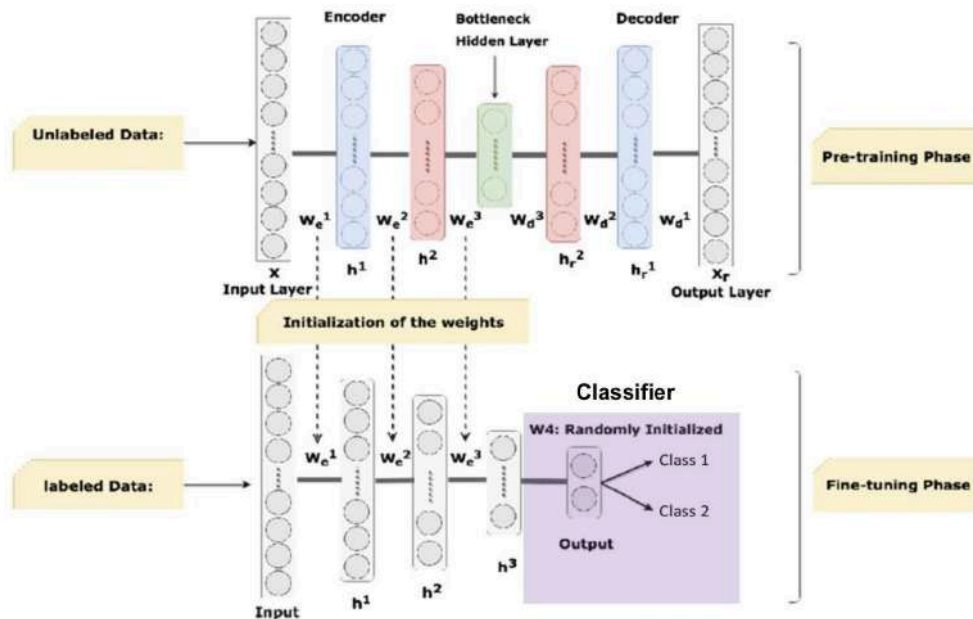
def decode(self, e):
    return self.decode(e)

```

Pre-training with Autoencoders

17. Pre-training

- Autoencoders can achieve similar results as transfer learning by pretraining on large set of unlabeled data, same type of data, just missing labels
- First train, satisfied, remove decoder with our own decoder



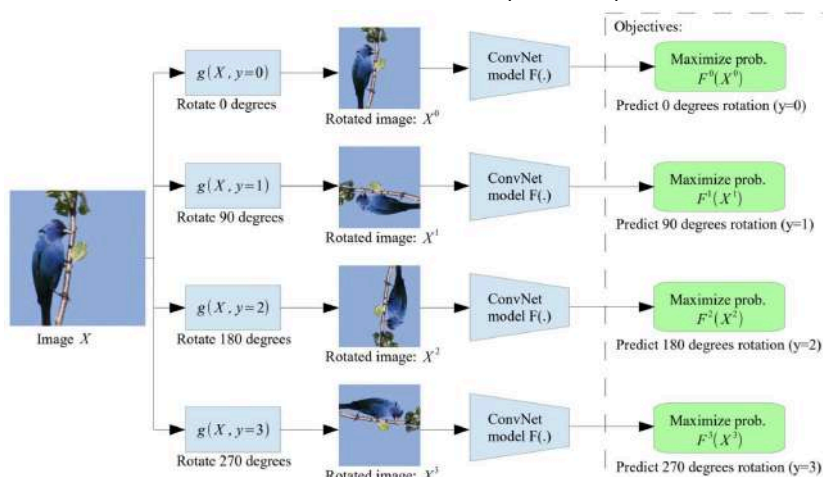
Self-Supervised Learning

18. Self-supervised learning with pretext tasks

- Proxy supervised tasks
 - The labels are generated automatically for free
 - Solving the task, requires the model to understand the content
- The challenge:
 - Devising the tasks such that they enforce the model to learn robust representations

19. RotNet

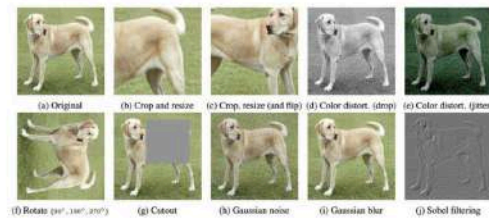
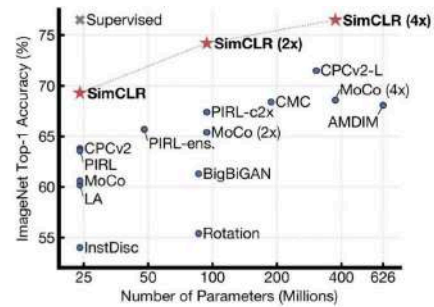
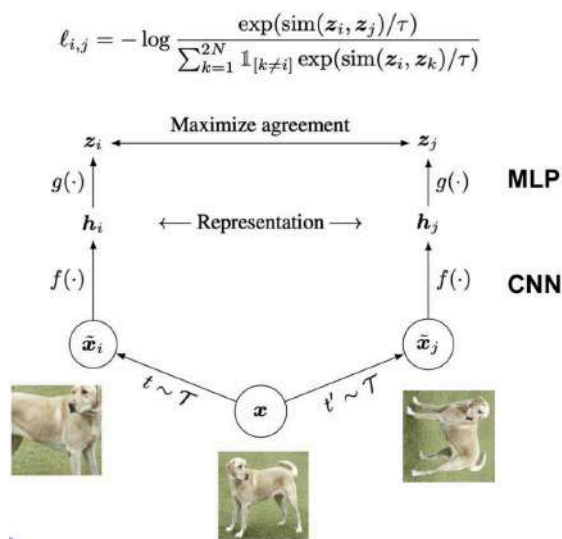
- Idea: Rotate images and make the model to predict the rotation angle
- Multiclass classification with 4 classes (CE loss) with free labels being generated automatically



20. Contrastive Learning

- Autoencoding methods
 - Reconstruct input
 - Compute the loss in output space
 - Compress all the details
- Contrastive methods
 - Contrast pair of positive/negative samples
 - Compute the loss in embedding space
 - Compress relevant information
 - Requires lots of negative examples

21. SimCLR



Convolutional Neural Network

Motivation

1. Inductive reasoning:
 - Start with an observation, leads to a possible generalization hypothesis. Valid observation may lead to different hypotheses, some of them can be false.
2. Inductive bias
 - The prior knowledge that you incorporate in the learning process that biases the learning algorithm to choose from specific functions might result in over-fitting and learning infeasible. It is any type of bias that a learning algorithm introduces in order to provide a prediction.
 - CNNs and inductive bias: architecture-specific biases that mostly depend on data and training procedure, mostly are locality and weight sharing translation invariance with pooling layers, translation equivariant without them being used.
3. Downsides for using a large fully connected network
 - Computation complexity grows: harder to train
 - Larger capacity: more data to generalize
 - Bad inductive bias: ignores geometry of image data
 - Good inductive bias in this case: use all information of the image, instead of only using some of them
 - Not flexible: Different image sizes require different models
 - 20*20 to 21*21, the size is bigger so the neural network needs to be scratched

Convolution Operator

4. Convolution
 - It is a mathematical operation on two functions f and g (one is the input function, the other one is the kernel), that expresses how the shape of one is modified by the other.

$$(f * g)[n] = \sum_{k=-\infty}^{\infty} f[k] g[n - k]$$

5. Convolution in 2D for images

- Computation

Convolution of Image I with filter kernel K

1. Multiply each pixel in range of kernel by the corresponding element of kernel
2. Sum all these products and write to a new 2d array
3. Slide kernel across all areas of the image until you reach the ends.

$$y[m, n] = I[m, n] * K[m, n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} I[i, j] \cdot K[m - i, n - j]$$

Put the kernel at the top-left corner of the image

7	2	3	3	8
4	5	3	8	4
3	3	2	8	4
2	8	7	2	7
5	4	4	5	4

*

1	0	-1
1	0	-1
1	0	-1

=

6		

$7 \times 1 + 4 \times 1 + 3 \times 1 + 2 \times 0 + 5 \times 0 + 3 \times 0 + 3 \times -1 + 3 \times -1 + 2 \times -1 = 6$

UNIVERSITY OF TORONTO *Top-left → one col right (r1) reach the ends → one row down → go to left → repeat (one col right)*

- Multiply by a fraction
 - Blurring averages out pixel intensities in an image

$$\star \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- The middle vertical line of a kernel all zeros
 - Vertical edge detector

$$* \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

- The middle horizontal line of a kernel all zeros
 - Horizontal edge detector

$$* \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- Another specific kernel
 - Blob detector: regions that differ in properties, such as brightness or color, compared to surrounding regions

$$* \begin{pmatrix} 0 & 0 & 3 & 2 & 2 & 2 & 3 & 0 & 0 \\ 0 & 2 & 3 & 5 & 5 & 5 & 3 & 2 & 0 \\ 3 & 3 & 5 & 3 & 0 & 3 & 5 & 3 & 3 \\ 2 & 5 & 3 & -12 & -23 & -12 & 3 & 5 & 2 \\ 2 & 5 & 0 & -23 & -40 & -23 & 0 & 5 & 2 \\ 2 & 5 & 3 & -12 & -23 & -12 & 3 & 5 & 2 \\ 3 & 3 & 5 & 3 & 0 & 3 & 5 & 3 & 3 \\ 0 & 2 & 3 & 5 & 5 & 5 & 3 & 2 & 0 \\ 0 & 0 & 3 & 2 & 2 & 2 & 3 & 0 & 0 \end{pmatrix}$$

- Kernels
 - Hand-crafted
 - Classic computer vision -> multi-stage feature (kernel) engineering
Character filtering, character segmentation, character recognition
 - Because you are randomizing the values initially, each kernel will convert to a different type of feature extractor. If you don't randomize initialization, you will have a very high chance of learning the same feature extractor across all the currents.

Convolutional Neural Networks

6. Biological Influence

- Hubel and Wiesel Cat Experiments (1958-1959)
 - Individual neurons respond to stimuli only in a restricted region of the visual field known as the receptive field
 - Collection of such fields overlaps to cover the entire visual area
 - Some neurons react only to images of horizontal lines, while others react line orientations
 - Higher-level neurons are based on the outputs of neighboring lower-level neurons (High-level from low-level)

7. Detecting:

- The output (activation) is high if the feature is present

8. Feature:

- something in the image, like an edge, blob, or shape

9. Convolutions with learned kernels:

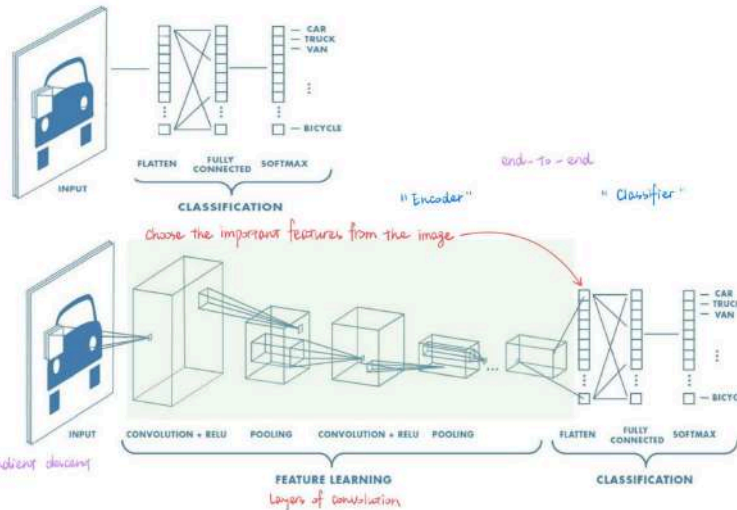
- share the same parameters across different locations (assuming input is stationary).

10. Characteristics

- Notion of proximity: logical correlations at the pixel-level
- A set of CNN kernels has less weight than a fully connected neural network that spans the entire image (weight sharing)
 - Weight sharing: Have several small kernels and connect all these outputs of the kernels to the neurons

11. CNNs

MLP → Requires that we preprocess the input!



CNN → Apply convolution to image tensors.

12. Forward and Backward pass

- Initialize the kernels randomly
- Forward pass: Convolve the image with the kernel
- Backward pass: update the kernel using gradients
- Initially randomly assigned numbers to kernel let the gradient descent update them

13. Zero Padding

- Adding zeros around the border of the image before convolution:
 - Keep width and height consistent with the previous layer
 - Keep the information around the border of the image

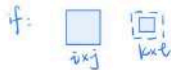
14. Stride

- Distance between two consecutive positions of the kernel:
 - Allows us to control the output resolution
 - e.g.: set the stride to 2 so the kernel skip two columns each time

15. Computing the output size

For each dimension of an input image with

- Image dimension of size i
- Kernel of size k
- Padding of size p
- Stride of size s



Need to apply formula (A) for each dim separately

$$O_1 = \lfloor (i + 2p - k) / s \rfloor + 1$$

$$O_2 = \lfloor (j + 2p - l) / s \rfloor + 1$$

The size of output dimension is computed by:

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1 \quad (A)$$



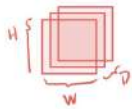
It assumes we kernel and input are square

16. Convolutional Neural Networks (ConvNets or CNNs)

- Reason for introducing convolutional filters into neural networks:
 - We don't have to handcraft the features
- Locally connected layers: local features in small regions of the image
- Weight sharing: detect the same local features across the entire image
- Neural network learns the kernel values (or weights)

17. CNN on RGB

- If we have three color channels



RGB

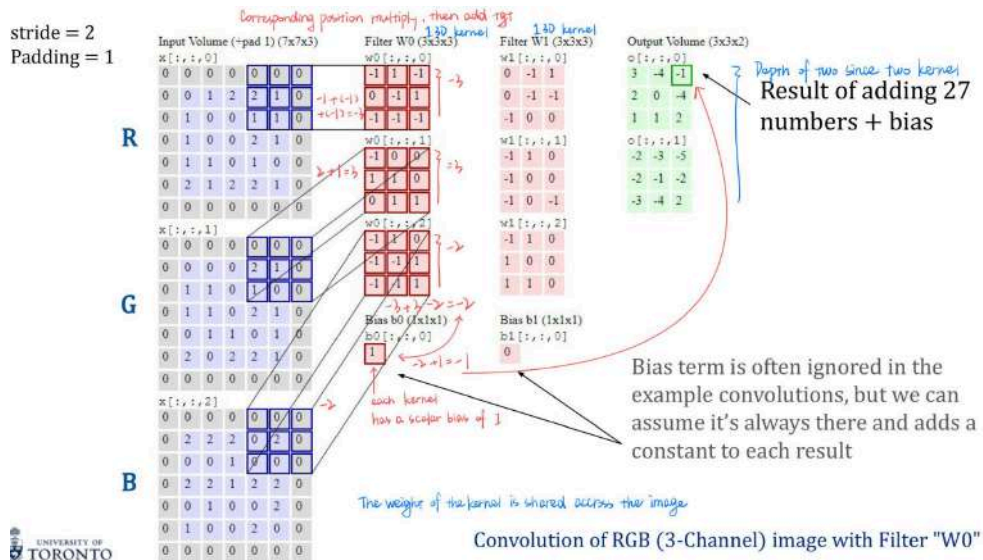
One number one value per pixel, intensity [0, 255]
Three values instead one per pixel for color image

18. Convolution on RGB input

- The depth of the kernel must match the depth of the image
- Element-wise multiplication: output will always has a depth of one

19. Detect multiple features

- Apply multiple kernels to the image in parallel at the same time, each kernel will give you a specific feature.
- The depth of the output = # of kernels you applied at the same time
- example
 - Pink values are weights learned by convolutional layer, everything else is input/output



20. Convolution on RGB input example

Colour input image: $3 \times 28 \times 28$

Convolution kernels: $5 \times 3 \times 8 \times 8$

Questions

- How many input channels are there? *3 (depth)*
- How many output channels are there? *5 (1x5)*
- How many trainable weights are there? *$5 \times 3 \times 8 \times 8 + 5 \text{ bias}$ (1 for each)*

Pooling Operator (something that reduces resolution)

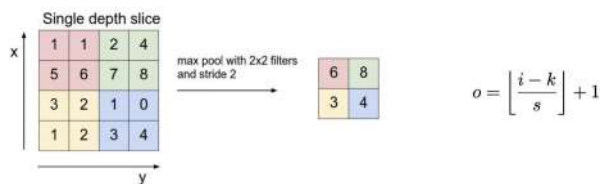
21. Consolidating information

- In a neural network with fully connected layers, we reduced the number of units before the final output layer
 - Because we want to consolidate information by compressing them (only remain the main information). It is the function of pooling layers
 - Consolidate information in a neural network with convolutional layers by:
 - ~strided convolutions
 - ~max pooling
 - ~average pooling

22. Max pooling (High-pass filter)

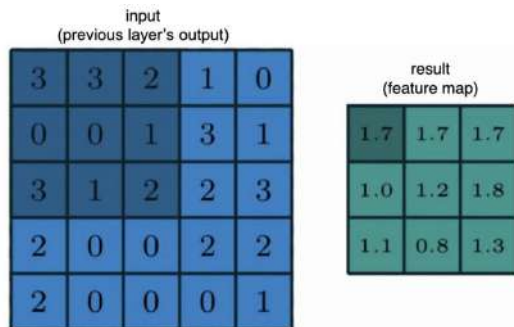
- Pooling layers provide invariance to small translations of the input

- The kernel won't have values within originally, they will just pick the max number within that size.



23. Average pooling:

- Compute the average value as the selected value



Max pooling generally works better

24. Stride convolution:

- Shift the kernel by s (e.g. s = 2) when computing convolution

25. CNN Architecture Blueprint

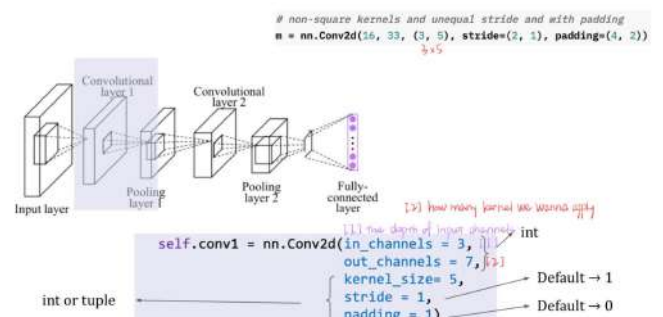
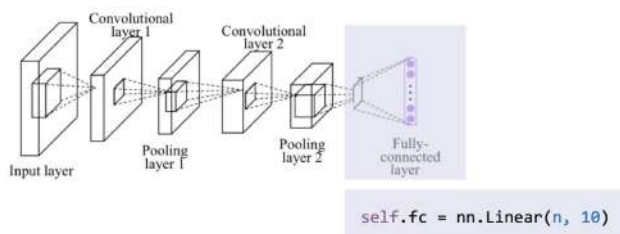
- As we go deeper, the resolution or the heights and width are reduced:
 - The kernel can capture a very small amount of information, if the height and width of the image are reduced, the same kernel can capture more information at once, but the resolution of the image will decrease since the size of the image is smaller.
- Why increase the # of kernels as going deeper:
 - The kernel that is closer to the input tends to learn low-level features, because there is a hierarchical nature to the information, we don't have many low-level features. Higher-level features are different combinations of low-level features, so many more possibilities.



Pytorch implementation

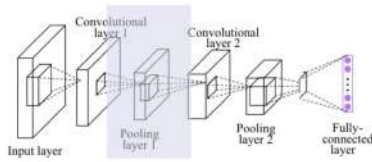
CNN in PyTorch: Conv2D

CNN in PyTorch: Recall Linear



CNN in PyTorch

CNN in PyTorch: MaxPool2d



```
self.pool = nn.MaxPool2d(kernel_size = 2,
                           stride = 2)
```

UNIVERSITY OF

```
class LargeNet(nn.Module):
    def __init__(self):
        super(LargeNet, self).__init__()
        self.name = "large"
        self.conv1 = nn.Conv2d(3, 5, 5) # 1 conv layer
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(5, 10, 5) # 2 conv layers
        self.fc1 = nn.Linear(10 * 5 * 5, 32)
        self.fc2 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # Convolutional layers
        x = self.pool(F.relu(self.conv2(x))) # Encoder
        x = x.view(-1, 10 * 5 * 5) # flattening it
        x = F.relu(self.fc1(x)) # 32 units
        x = self.fc2(x) # 10 units
        return x
```

Handwritten notes:
 - Pooling is non-parametric layer (has no bias and weights)
 - and-to-end learning (Don't need to manually do anything between)
 - The loss func compute changed weight and the gradient deliver upwards one by one.

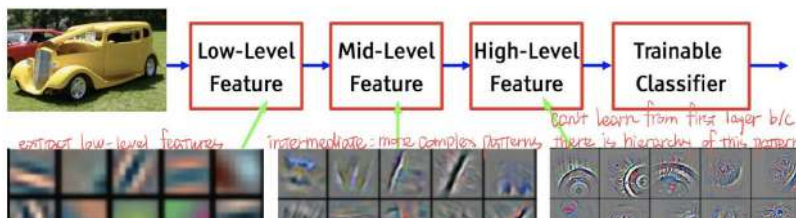
Visualizing convolutional filters

26. CNN filters/feature maps look like

- The first layer is usually to do all the research that people usually to do manually
- Learning with different edges
- The output is the shadow of the image

27. CNNs learn what features

- Need to build features on top of features until you get to this high-level layer



Saliency maps: Use gradients of the output over the input to highlight the areas of the images which are relevant for the classification. Similar to the heat map

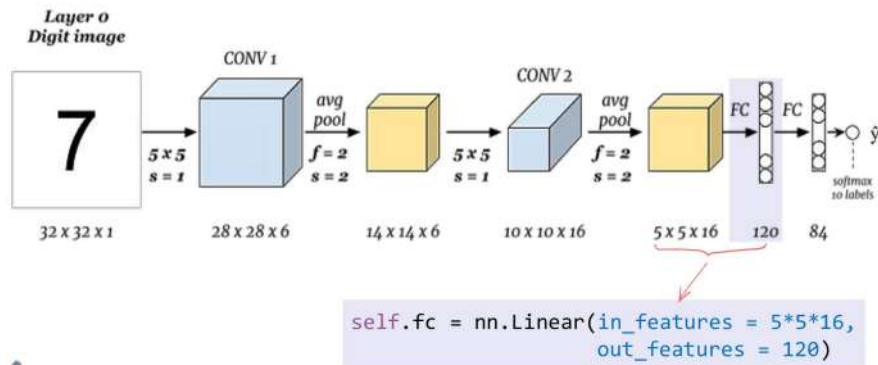
1. Feed the image to the network
2. Compute the gradients back to the input image
3. Take the maximum value of absolute gradients across channels
4. Visualize

Unfortunately outside giving some intuition, these are not practically very useful, and sometimes even misleading

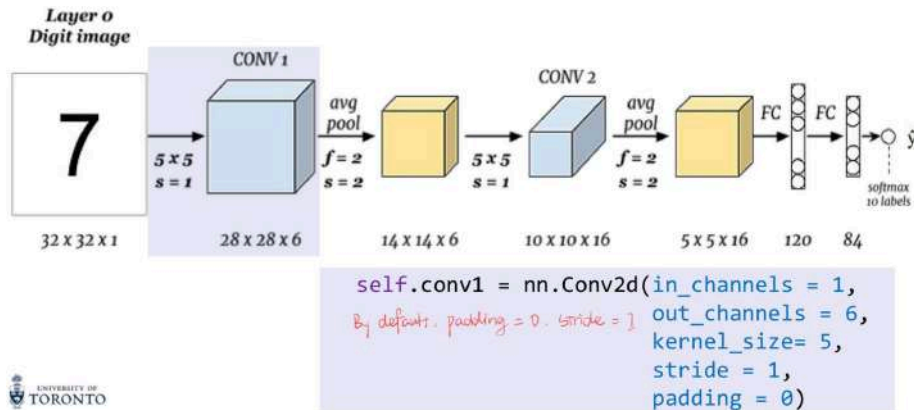
CNNs in Pre-Deep Learning Era

28. LeNet

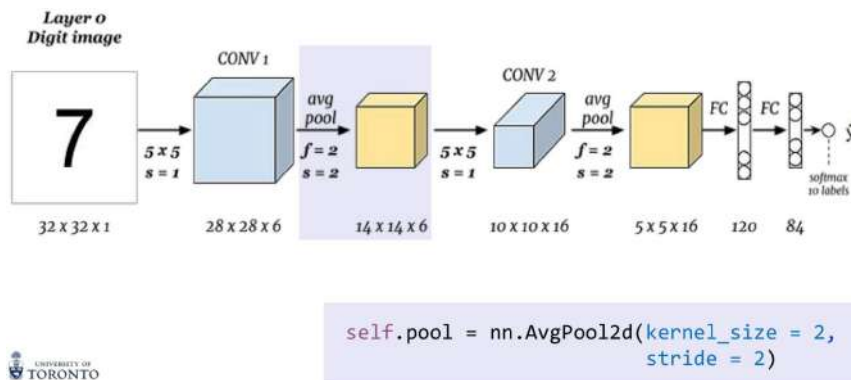
- The original CNN, first introduced by Yann LeCun in 1989
 - Based on earlier "Neocognitron" (Fukushima, 1980)
- Several variants, mostly referred to LeNet-5 (above, 1998)
 - 7 layers total: 2 convolutional, 2 subsampling (i.e. pooling), 3 fully-connected
- Different invariance:
 - Translation invariance
 - Scale invariance
 - Rotation invariance
 - Squeeze/stretch invariance
 - Stroke width invariance
 - Noise invariance
- Pytorch implementation
 - Fully-connected layersc



- Convolutional layers



- pooling/subsampling layers



```
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(5 * 5 * 16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 5 * 5 * 16)
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        x = self.fc3(x)
        return x
```

Activation func but with problem with sigmoid saturation

LeNet-5 (modern version)

29. On the eve of deep learning

- Visual object classification
- Deformable Parts Models <- best resolutions to object classification

- CNNs are outperformed on most tasks by using hand-crafted computer vision features, and other ML classifiers, e.g. random forests (decision trees) or SVM

30. Deformable Parts Models

- Recognize using parts and locations of parts
- Allow some deformation of part location
- Doesn't work well for different viewing angles

Modern Architectures

31. ImageNet

ImageNet Large Scale Visual Recognition Challenge

- Pascal VOC was ~20,000 images with 20 classes (2006 - 2009)
- ImageNet was the first large-scale image dataset (14 million images)
- ILSVRC dataset based on ImageNet
 - 1 Million training images
 - 1000 different classes!
 - 50k validation, **test set (never released)**
- When we say ImageNet we mean ILSVRC

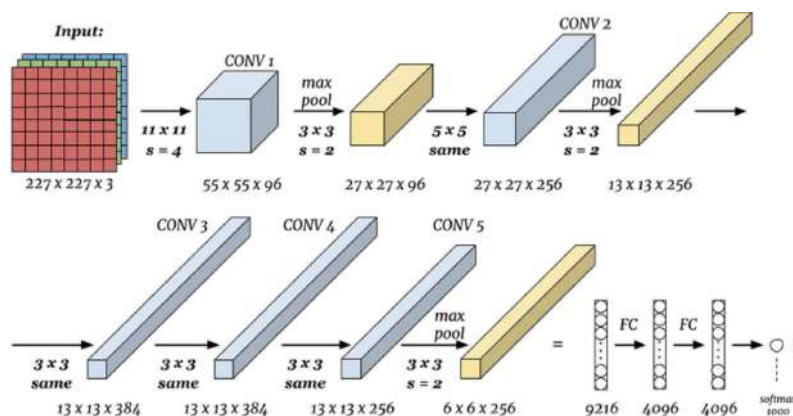


32. AlexNet

- General
 - ILSVRC challenge ran from 2010.
 - Like Pascal VOC, every year saw the new winner improve accuracy by ~1-2%
 - CNN entry (AlexNet) in 2012 improved accuracy over previous year by ~10%
 - This is when "Deep Learning" began

Model	Top-1	Top-5
Sparse coding [2]	47.1%	28.2%
SIFT + FVs [24]	45.7%	25.7%
CNN	37.5%	17.0%

- Architecture



- Different from LeNet-5

Deep Learning is differentiated from vanilla **Neural Networks** mostly in the changes between LeNet-5 and AlexNet:

- Much **larger training datasets** (e.g. ImageNet)
- Vast **increase in compute/GPU acceleration** (imagine 1989 PC v.s. 2012!)
- Much **larger model size/more layers**, enabled by both of the above!

AlexNet Training/Architecture Improvements:

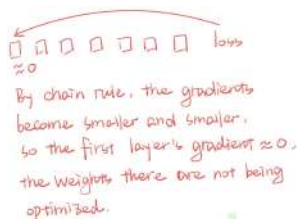
- Large number of convolutional layers (i.e. deeper model)
- Use ReLU activation functions instead of sigmoids
- Dropout, **data augmentation**
- training
 - ~60 Million parameters!
 - Used GPUs to accelerate compute: 2 Nvidia GTX 580 GPUs
 - 5-6 days to train over 90 epochs
 - Optimized with SGD + Momentum
 - Uses **weight decay**, **dropout** & **data augmentation** to improve generalization
 - Learning rate schedule decreased learning rate 3 times over training

33. Data Augmentation

- Apply class-preserving transformations to the input
 - Increases training data
 - Helps generalization by learning the internal representation of transformations
- Used by AlexNet (and all other CNNs)
- Generally, different positions, colors, and directions of the same picture

34. Generalization and Depth

- Increased depth improved generalization on ILSVRC and other tasks, but training very deep models often failed:
 - Vanishing or exploding gradients: Gradients will get smaller and smaller if you get many small numbers in these intermediate layers
- improvements:
 - Improved initialization for ReLUs
 - Normalization (e.g. Batch Normalization)
 - Residual connections



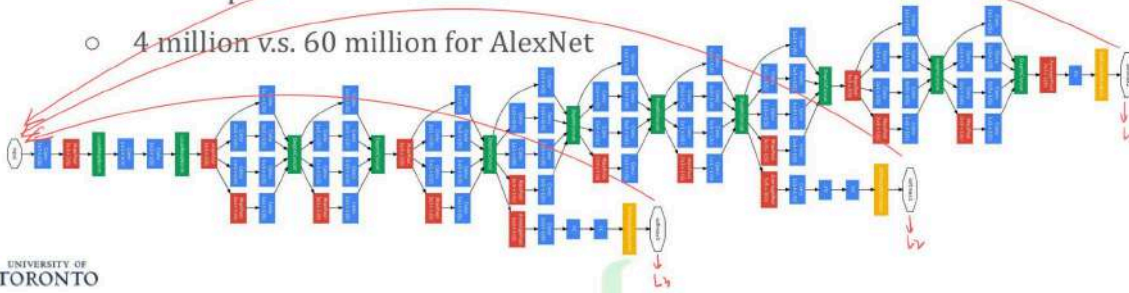
35. New model to solve this problem

L1 compute all, L2 - L3 compute partial to avoid vanishing gradients

$L = L_1 + L_2 + L_3 \Rightarrow (\partial L / \partial w_1) + (\partial L / \partial w_2) + (\partial L / \partial w_3)$

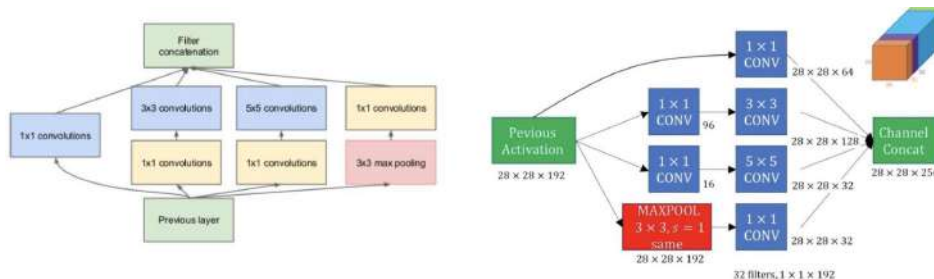
GoogLeNet (Inception)

- 2014 ILSVRC winner, 6.67% Top-5 error
 - Human gets ~5.1%
- Primary motivation was to go deeper
 - 22 convolutional layers
- Much more parameter efficient than AlexNet
 - 4 million v.s. 60 million for AlexNet



36. Inception block

- Use 1*1 layers to compute depth, so the consecutive convolutional kernel has less depth, so smaller number of parameters to be optimized
- Uses a mixture of 3*3, 5*5 and 7*7 filters on one layer
- Don't need large 7*7 to learn most important filters, use mostly 3*3, and add a few larger filters



37. Pointwise (1*1) convolution

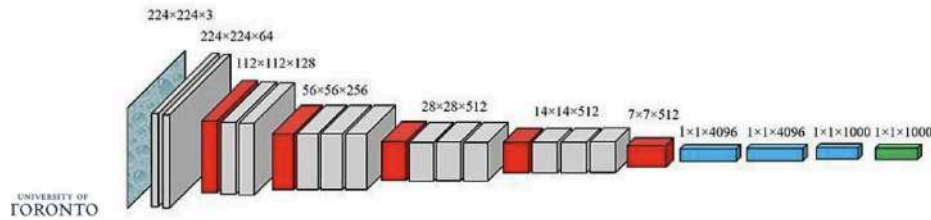
- Control the depth of your network in different layers
- Pixel-wise linear transformations
 - Originally used in "Network-in-Network" model
- Learn to map CNN feature maps into a lower or higher dimensional space
 - Good for learning compact representations/compression
- Used in all modern CNN architectures, except VGG

38. Auxiliary Loss

- Address the gradient vanishing problem -> introduce intermediate loss function
- Inception network is pretty deep -> subject to the vanishing gradient problem
- Solution -> intermediate classifiers
 - Adding classifiers in the intermediate layers such that the final loss is a combination of the intermediate losses and the final loss

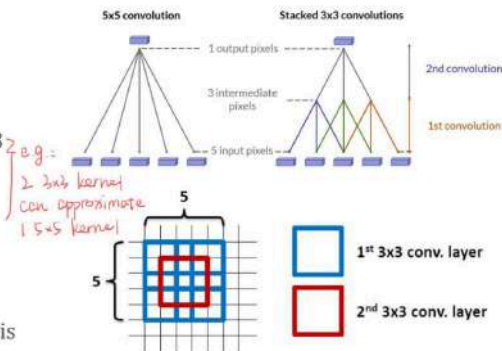
39. VGG (Visual Geometry Group, Oxford)

- 2014 ILSVRC classification 2nd place, 7.3% Top-5 error
 - However, won parallel ILSVRC localization challenge
- Proposed Models with 11, 13, 16, and 19 layers
- Very simple architecture, easy to understand/extend
- Very large number of parameters: 138 Million v.s. 60 Million for AlexNet!

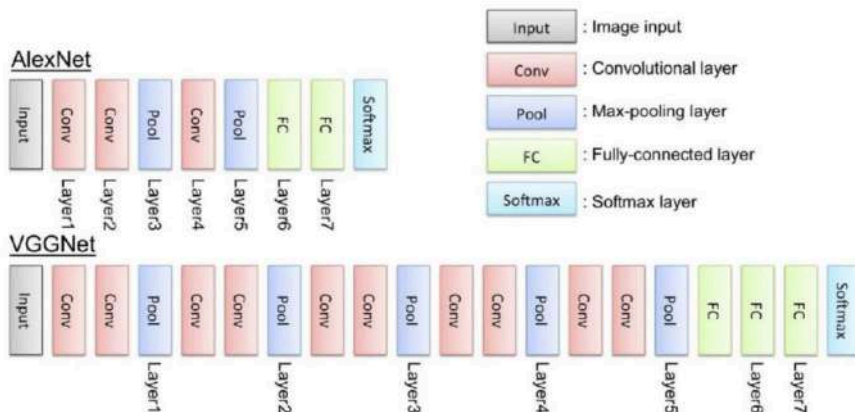


VGG was a very impactful paper:

- Simple architecture made of simple stacked blocks
- We only need **3x3 filters**
 - Authors pointed out that stacked 3x3 filters can approximate any larger-sized convolution, more efficiently
 - Since VGG almost all CNNs use mostly/exclusively 3x3 filters!
 - The data augmentation used by VGG is very commonly used



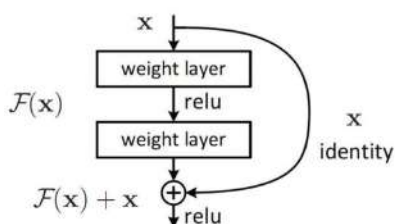
AlexNet v.s. VGG



40. Residual Networks

- Uses skip connections to provide deeper layers more direct access to signals
- ResNet won ILSVRC 2015 with 3.57% error
 - The model had 152 layers
 - Better than human baseline

41. Skip Connections (residual Networks)



normal layer:

`next_activation = layer(activation)`

residual layer

`next_activation = activation + layer(activation)`

42. ResNets

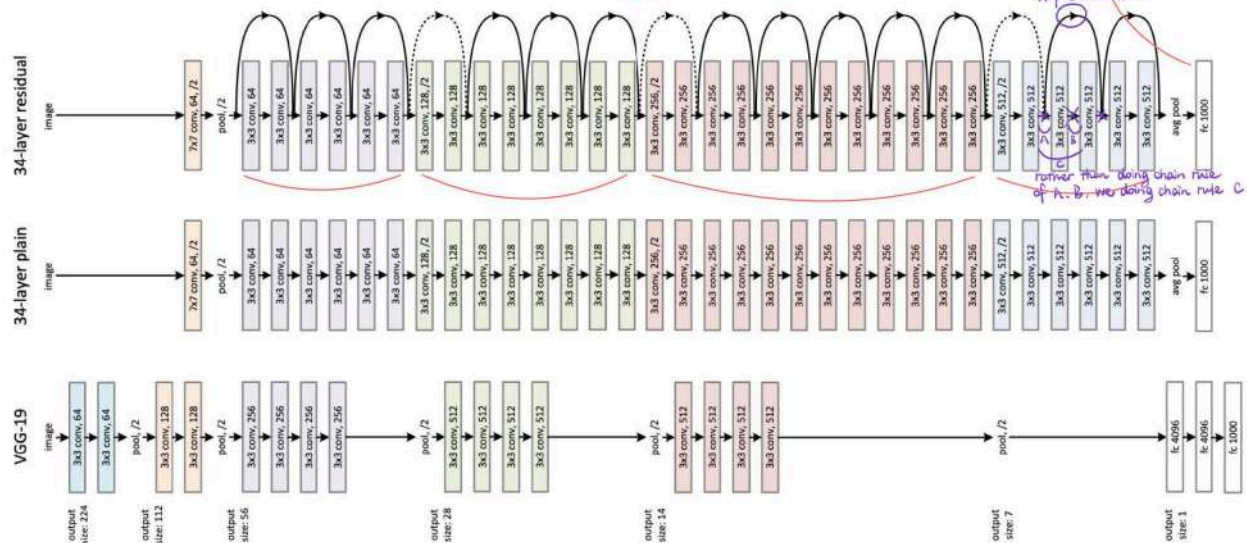
If we do residual connections, it do "total gradient = gradient from original path + new connection g". Global avg pooling get the avg; each block become one number

Residual Networks (ResNets)

since size is not the same, we choose to apply a linear layer to match the size

Don't have classifiers

skip connection

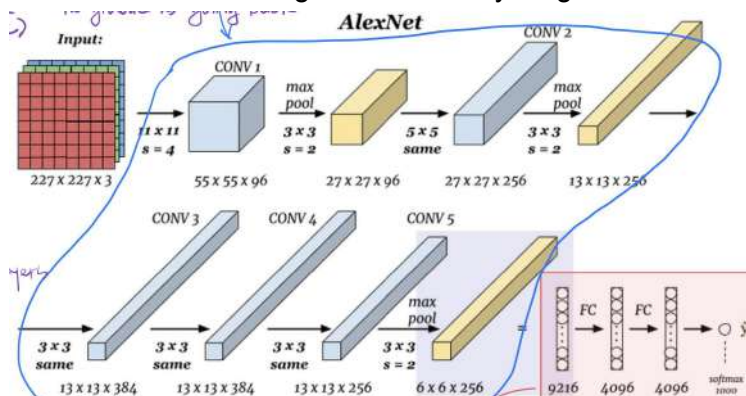


- Residual blocks (multiple convolutions with **skip connections**)
- Downsampling using stride 2, instead of max/avg pooling
- Global average pooling after last convolutional layers (introduced by Network-in-Network)
 - Means that the embedding has no spatial dimension and is only 512 floats!
- Only a **single** fully-connected **classification layer**
 - learned embeddings are so good we don't need a complex classifier at end of model

Transfer Learning

43. Learning Visual Features

- Encoder is responsible for extracting important features of an input
- If we want it to fit our small dataset:
 - Delete the original classifier, freeze the weights in encoding part (stop the gradient there, no gradient going back, only updating weights of the two fully connected layers)
- Classifier steps: Map these features to the labels that you have for that past.
- Two distinct parts:
 - Convolutional layers: Learn filters across spatial and channel dimensions
 - Fully-connected layers: learn to classify images based on the learned visual features
- Embedding: a learned lower-dimensional set of "visual features" representing the image
 - This embedding encodes everything needed from the image to classify objects



44. Transfer Learning using Embeddings

- By being trained on a large image classification dataset, CNNs learn something general about representing images.
- We use these features to transfer our learning to a new problem:
 - Train CNN (e.g. AlexNet) on large image datasets (e.g. ImageNet)
 - Remove “classification” layers at end of model, freeze remaining weights
 - Add, and train, new layers at end of model suitable for our new task

45. Fine-tuning for transfer learning

- We froze the original model’s weights, used our CNN layers as a feature extractor
- Often training some/all of the original model’s weights on the new task at a lower learning rate helps the features “adapt” to the new task

46. PyTorch implementation

- All of the models we've discussed (and more!) are available in torchvision
- We can avoid the **large computation** needed for training a state-of-the-art model, and just use pre-trained models
- You can also train the models from scratch with → pretrained=False
- Keep this in mind for your projects!

```
import torchvision.models
```

```
alexnet= torchvision.models.alexnet(pretrained=True)
```

```
Inception=  
torchvision.models.inception.inception_v3(pretrained=True)
```

```
vgg16= torchvision.models.vgg.vgg16(pretrained=True)  
vgg19= torchvision.models.vgg.vgg19(pretrained=True)
```

```
resnet18=  
torchvision.models.resnet.resnet18(pretrained=True)
```

```
resnet152=  
torchvision.models.resnet.resnet152(pretrained=True)
```

```
feature_data = alexnet(image)
```

Artificial Neural Networks

Neuron

1. General

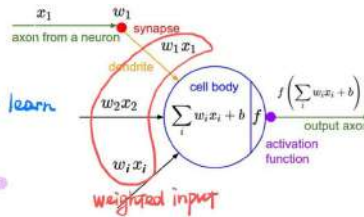
x_i is the **input** such as a pixel in an image

w_i is the **weight** for input x_i that we learn for this particular input *weight we assign to that input*
It is what we're gonna use to learn

b is the **bias**, a **weight we learn with no input**

f is the **activation function** that **determines how our output changes with the sum of all weight-input products**

y is the **output** such as the class an image belongs to



Activation Function

2. Activation function

- Good to develop our own activation function
 - It needs to have lightweight derivatives -> computationally cheap
 - Suggestion: use ReLU as the default activation function

3. Linear Activation Function

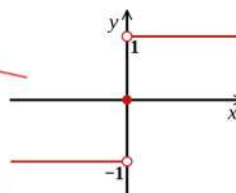
- Bias term: if we don't have it, the decision boundary will always pass through the origin which is kinda limited.
- The neural network automatically updates the bias term
- The bias is related to the offset of the line from the origin
- Problems
 - Most real datasets are not linearly separable
- $(y = wx + b)$ is a generalized line for any dimension, known as a hyperplane, splitting the n -dimensional input space into 2
 - Given you the decision boundary

4. Early Activation Functions: Perceptrons

First artificial neurons (1943-70s) used a simple binary activation function based on which side of the hyperplane the input is:

$$f(x) = \text{sign}(x) \quad \text{Sign function}$$

$$f(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases} \quad \text{Heaviside (unit) step function}$$



if a func is mostly differentiable or mostly continuous, we can use it.
This is called the **decision boundary**
means that we cannot apply efficient optimization techniques that we use in this function to dysfunction.
These functions are not differentiable, continuous, or smooth

5. Sigmoid Activation Function

- Saturated neurons "kill" the gradients
 - Gradients become vanishingly small very quickly away from $x = 0$
 - The functions get saturated (gradients = 0), we will not be able to use it, since we don't have any signal to train it

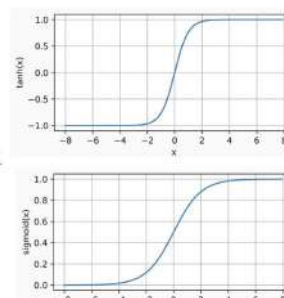
Sigmoid activation functions were the most common before 2012:

- Easily differentiable, smooth, continuous
- Range between $[-1, 1]$ or $[0, 1]$

There are many sigmoid functions, the most common are:

$$f(x) = \tanh(x) \quad \text{Hyperbolic tangent} \quad \text{Between } -1 \text{ and } 1$$

$$f(x) = \frac{1}{1 + e^{-x}} \quad \text{Logistic function} \quad \text{Between } 0 \text{ and } 1$$



6. ReLU Activation Function

- Problem with original ReLU (Rectified Linear Unit) based activation functions:
 - Lose half of the information on the negative side
- For Parametric ReLU and Leaky ReLU, the negative slope is being learned by your neural network
- Can the negative slope be one?
 - No. We can have any value except 1. Otherwise the function will be linear again.

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

ReLU

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative_slope} \times x, & \text{otherwise} \end{cases}$$

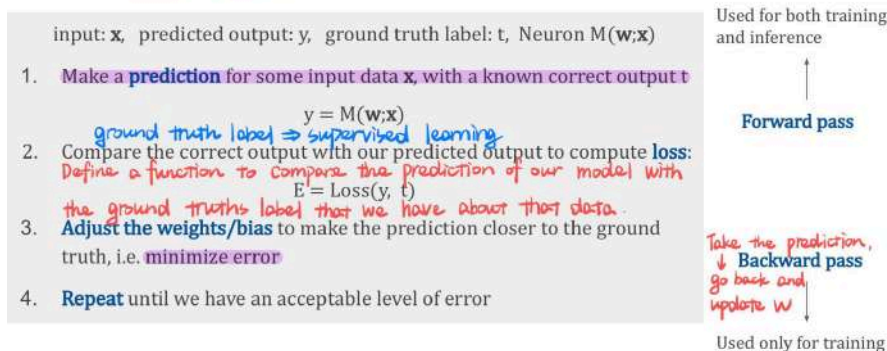
Leaky ReLU

$$\text{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ ax, & \text{otherwise} \end{cases}$$

Parametric ReLU

Training Neural Networks

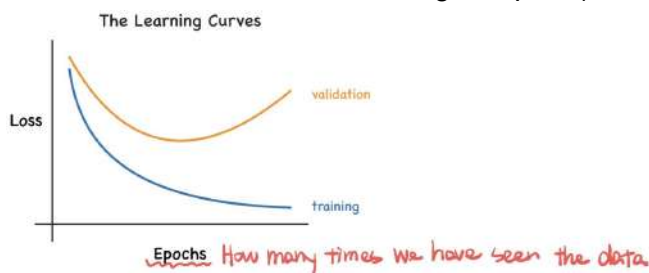
How do we **learn** the **weights** (and bias) of a neural network?



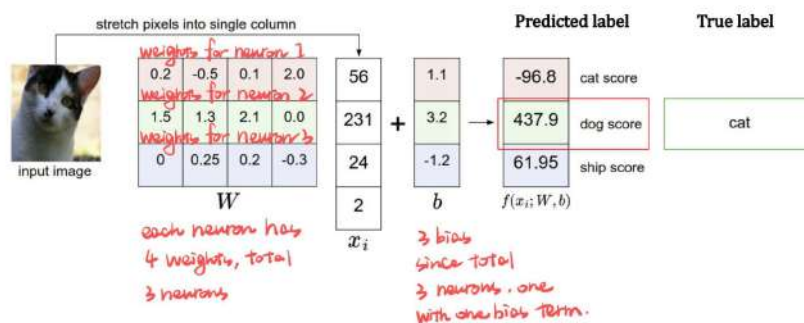
Loss Function

7. Loss function:

- computes how bad predictions are compared to the ground truth labels
- Large loss: the network's prediction differs from the ground truth
- Small loss: the network's prediction matches the ground truth
- Calculate the error over all training samples (average error)



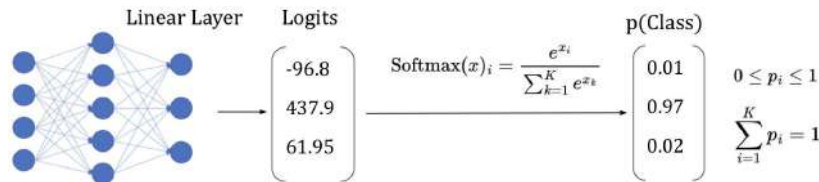
Suppose we want to train a linear neuron to differentiate images into three classes:



8. Softmax function:

- normalizes the logits into a categorical probability distribution over all possible classes
- Ground-truth label: Human-defined identification since we trust human's classification

- One-hot encoding: Maps categories to vector representation
- Softmax function itself: the exponential of each of the inputs divided by the summation of the exponential of all the inputs.



9. Mean Squared Error (MSE):

- mostly used for regression problems

$$\text{MSE} = \frac{1}{N} \sum_{n=1}^N (y_n - t_n)^2$$

Number of training samples

Prediction

True label

Predicted p(Class)	Ground truth
0.01	1.0
0.97	0.0
0.02	0.0

$$\text{MSE} = (0.01 - 1.0)^2 = 0.98$$

10. Cross Entropy (CE):

- mostly used for classification problems

$$\text{CE} = -\frac{1}{N} \sum_{n=1}^N \sum_{k=1}^K t_{n,k} \log(y_{n,k})$$

training samples

#classes

True label

Prediction

Predicted p(Class)	Ground truth
0.01	1.0
0.97	0.0
0.02	0.0

historical issue of one base 2

$$\text{CE} = -[1.0 \times \log_2(0.01) + 0.0 \times \log_2(0.97) + 0.0 \times \log_2(0.02)] = 6.64$$

11. Binary cross entropy (BCE)

$$\text{BCE} = -\frac{1}{N} \sum_{n=1}^N [t_n \log(y_n) + (1 - t_n) \log(1 - y_n)]$$

12. Forward-Pass with Error Calculations

- MSE

```
import math

x = [[1.0, 0.1, -0.2], # data N=4
     [1.0, -0.1, 0.9], # 4 examples
     [1.0, 1.2, 0.1],
     [1.0, 1.1, 1.5]]
t = [0.0, 0.0, 0.0, 1.0] # labels (ground-truth labels)
w = [1, -1, 1] # initial weights

def simple ANN(x, w, t):
    total_e, e, y = 0, [], []
    for n in range(len(x)): N=4
        v = 0
        for d in range(len(x[0])):
            v += x[n][d] * w[d] = sum(W_i x_i)
        y.append(1/(1+math.e**(-v)) # sigmoid
        e.append((y[n]-t[n])**2) # MSE
    total_e = sum(e)/len(x)
    return (y, w, total_e)
```

Compute the prediction

Compute the error

- BCE

```

import math

x = [[1.0, 0.1, -0.2], # data
      [1.0, -0.1, 0.9],
      [1.0, 1.2, 0.1],
      [1.0, 1.1, 1.5]]
t = [0, 0, 0, 1] # labels
w = [1, -1, 1] # initial weights

def simple_ANN(x, w, t):
    total_e, e, y = 0, [], []
    for n in range(len(x)):
        v = 0
        for d in range(len(x[0])):
            v += x[n][d] * w[d]
        y.append(1/(1+math.e**(-v))) # sigmoid
        e.append(-t[n]*math.log(y[n])-(1-t[n])*math.log(1-y[n])) # BCE
    total_e = sum(e)/len(x)
    return (y, w, total_e)

```

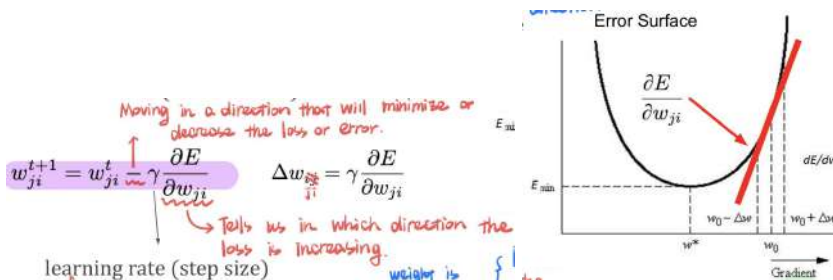
Gradient Descent (An algorithm from optimization)

13. Neural Network Layer (Vector, Matrices, Tensors)

- Weight matrix: Each neuron's weight vector is a row of the weight matrix W and the input is a column vector x
 - $Y = f(Wx + b)$
- How to change each of our neuron's weights w_{ji} to reduce this error E
 - Know how much each weight is contributing to the loss or error $\rightarrow dE/dw_{ji}$
 - We want to find the error to increase and then change them in the opposite direction
 - Relatively simple to calculate adjacent to the output layer

14. Neural Network Single-Layer Training

- Train the neural network: computing the gradient of loss with respect to each of the weights, then changing that weight based on the value that we compute if the weight is causing higher values of loss, we will move the opposite direction.
- Gradient: vector of partial derivatives for all weights
 - Direction of the gradient is the direction in which the function increases most quickly
 - Magnitude of the gradient is the rate of increase
- Learning rate: a fixed number to make sure we are not very aggressively changing the weights
- Adjusting weights according to the slope (gradient) will guide us the minimum (or maximum) error
- Weight is contributing positively: reducing the loss we are going to move in the positive direction of that.



15. Delta Rule for Single Weight/Training Sample

$E = (y - t)^2$ $f(x) = \frac{1}{1 + e^{-x}}$ Chain rule! $\frac{dE}{dw_p} = \left(\frac{dE}{dy} \right) \left(\frac{dy}{da} \right) \left(\frac{da}{dw_p} \right)$

$\frac{dE}{dy} = \left(\frac{d(y-t)^2}{dy} \right) = 2(y-t)$
 $\frac{dy}{da} = \left(\frac{d\left(\frac{1}{1+e^{-a}}\right)}{da} \right) = (1-y)(y)$
 $\frac{da}{dw_p} = x_p$
 $\frac{dE}{dw_p} = 2(x_p)((y-t)((1-y)(y)))$

16. Forward-pass and backward-pass

```
def simple_ANN(x, w, t, iter, lr):
    total_e = 0
    for i in range(iter):
        e, y = [], []
        for n in range(len(x)):
            v = 0
            for d in range(len(x[0])):
                v += x[n][d] * w[d]
            y.append(1/(1+math.e**(-v))) # sigmoid
            e.append((y[n]-t[n])**2) # MSE

        # gradient descent to update weights
        for p in range(len(w)):
            d = 2*x[n][p]*(y[n]-t[n])*(1-y[n])*y[n]
            w[p] -= lr*d
        total_e = sum(e)/len(x)
    return (y, w, e)
```

Neural Network Architectures

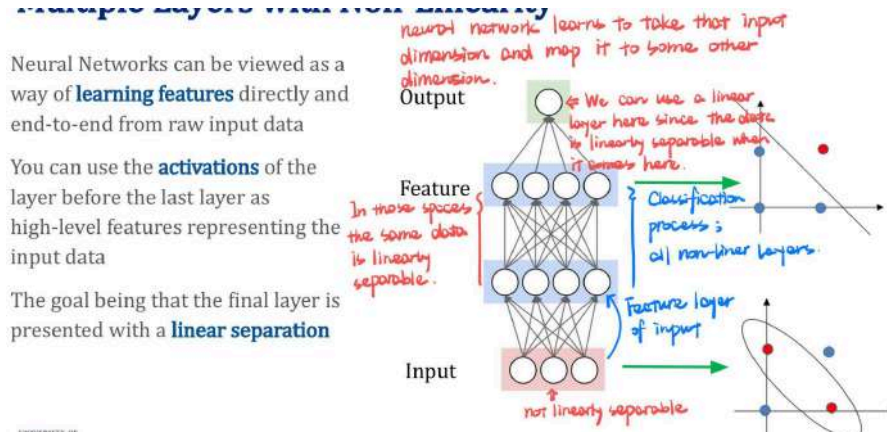
17. XOR

- Needs two decision boundaries to solve
- Solution:
 - Have at least one hidden neural network layer
 - Limit of an infinitely wide neural network with at least one hidden layer, NN is a universal function approximator

18. Backpropagation:

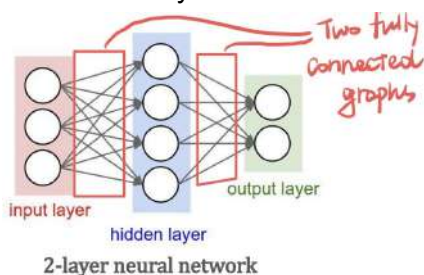
- Solving credit assignment problem
- A method that describes how to distribute errors to neurons not adjacent to the output layer
- Solution: Dynamic programming

19. Multiple Layers with Non-Linearity



20. Neural Network Architecture

- Feed-Forward Network: Information only flows forward from one layer to a later layer, from the input to the output.
- Fully-Connected Network: Neurons between adjacent are fully connected
- Number of Layers: number of hidden layers + output layer (Input layer is not the layer here)



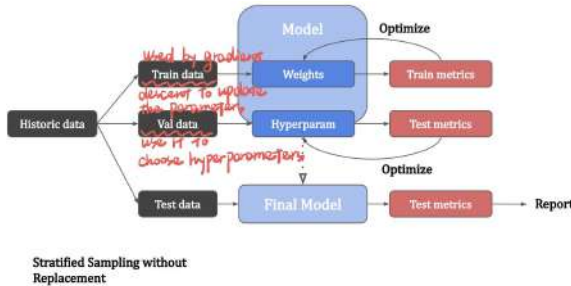
- An architecture of an NN describes the neurons and their connectivity.

Training Artificial Neural Networks

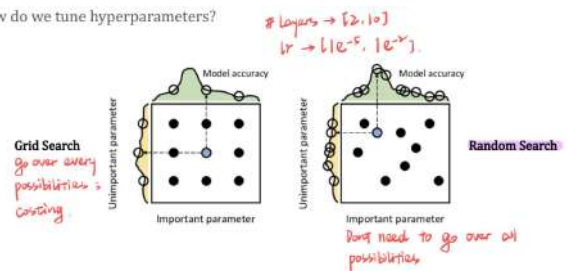
Hyperparameters

1. General

- Different hyperparameters:
 - Batch size
 - Number of layers
 - Layer size
 - Type of activation function
 - Learning rate
- Weights are updated through gradient descent (Inner loop of optimization)
- Tune hyperparameters (Outer loop of optimization)



How do we tune hyperparameters?



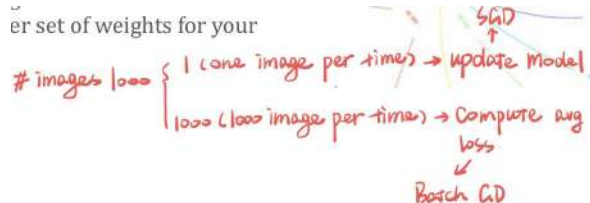
Optimizers

2. general

- Defining a loss function turns a learning problem into an optimization problem
- Optimizer:
 - Determines, based on the value of the loss function, how each parameter (weight) should change
 - Solves the credit assignment problem: how do we assign credit to the parameters based on how the network performs?
- PyTorch automates the gradient computation

3. Stochastic Gradient Descent (SGD)

- For each iteration evaluate a training sample from the dataset taken at random
- It allows you to do more of a global search for an optimum, results in a better set of weights for your model
- Gradient descent on entire training data



4. Mini-Batch Gradient Descent

- Advantages of applying batching:
- Batch size: number of training examples used per optimization “step”
 - Randomly select, Batch the data, train based on the batch
 - Often set the batch size to your gpu memory (number of images)
- Iteration: One step
 - The parameters are updated once per iteration
 - A number of samples were processed before the model is updated
- Epoch: number of times all the train data is used once to update the parameters

5. Inefficient batch size

- Too small:
 - Noisy
 - Optimize a possibly very different function loss at each iteration

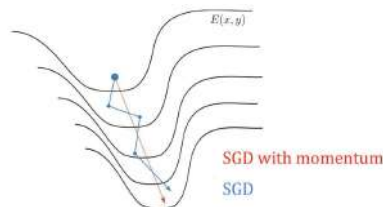
- Too large:
 - Expensive: need to do parallel computation
 - Average loss might not change very much as batch size grows
 - The true gradient is not always the best gradient for optimization
i.e. some amount of noise in your gradients can help training (converge faster), larger batch size is not always better

6. Gradient descent: N-Dimensional

- Plateaus are a problem but can be addressed using specialized variants on gradient descent
- Most points of zero gradients are saddle points
 - Saddle points: The gradients are zero in one direction, non-zero in the other directions.

7. SGD with Momentum

- Ravines: areas where the surface curves much more steeply in one dimension than in another, common around local optima.
- Problem:
 - Navigating ravines: It oscillates across the slopes of the ravine
- Why momentum:
 - Helps accelerate SGD in the relevant direction
 - Dampens oscillations



- Characteristics:
 - The momentum term increases for dimensions whose gradients point in the same directions
 - Reduces updates for dimensions whose gradients change directions
 - Analogy: We push a ball down the hill, ball becomes faster and faster until it reaches the terminal velocity

$$\lambda \left(\gamma \frac{\partial E}{\partial w_{ji}} \right)^{t-1} - \left(\gamma \frac{\partial E}{\partial w_{ji}} \right)^t \quad \left\{ \begin{array}{l} v_{ji}^t = \lambda v_{ji}^{t-1} - \gamma \frac{\partial E}{\partial w_{ji}} \\ w_{ji}^{t+1} = w_{ji}^t + v_{ji}^t \end{array} \right.$$

moving the avg along the gradient

Momentum Coefficient lr Velocity

8. Adaptive Moment Estimation (Adam)

- Adaptive learning rates: Each weight has its own rate

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \left(\frac{\partial E}{\partial w_{ji}} \right) \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial E}{\partial w_{ji}} \right)^2$$

momentum momentum gradient gradient

$$w_{ji}^{t+1} = w_{ji}^t - \frac{\gamma}{\sqrt{v_t} + \epsilon} m_t$$

in order to change the learning rate lr vary tiny number to avoid from zero

- Incorporates momentum and adaptive learning rate:
 - Rapid convergence requires minimal tuning
 - Commonly used optimizer

```
torch.optim.Adam(model.parameters(), lr=0.001)
```

10⁻³ ~ 10⁻⁴ to start

Learning Rate

9. Learning rate:

- determines the size of the step that an optimizer takes during each iteration

$$w_{ji}^{t+1} = w_{ji}^t - \gamma \frac{\partial E}{\partial w_{ji}}$$

- Larger step size: Make a bigger change in the parameters (weights) in each iteration
- Use small value:
 - Very small parameter change
 - Longer training time
- Use bigger value:
 - Noisy
 - Detrimental to training
- Appropriate Learning Rate depends on:
 - The learning problem itself
 - The optimizer
 - The batch size:
 - Large batch -> larger learning rates
 - Small batch -> smaller learning rate
 - The stage of training:
 - Reduce as training progresses

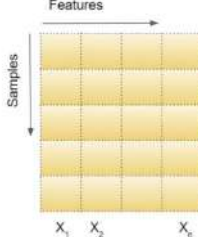
Normalization

10. Reason for normalization

- Prevent the model from paying attention to the features with larger range
- Normalize one layer:

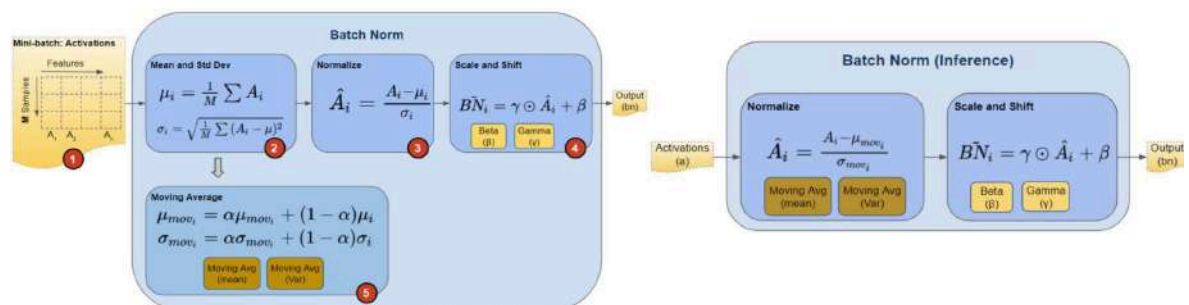
$$X_i = \frac{X_i - \mu_i}{\sigma_i}$$

mean squared



11. Batch Normalization

- Inference time
 - Keep a moving average during training and use it at inference time



- Normalize activations batch-wise for each layer

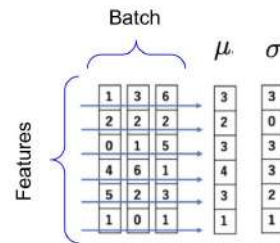
normalize the inputs

Input: Values of x over a mini-batch: $B = \{x_1 \dots x_m\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad // \text{ mini-batch variance}$$

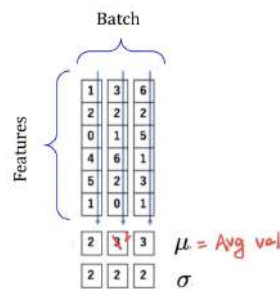
$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$


- **Advantages:**
 - Higher learning rate -> speed up the training
 - Regularize the model
 - Less sensitivity to initialization
- **Disadvantages:**
 - Depends on batch size: no effect with small batches
 - Cannot work with SGD <- only works with batch 1

12. Layer Normalization

- Applied on the neuron for a single instance across all features
- **Advantages:**
 - Simpler to implement, no moving averages or parameters
 - Not dependent on batch size



Regularization

13. Regularization:

- a set of techniques that you make the training task more difficult for the model.

14. Dropout:

- forces a neural network to learn more robust features
- During training: randomly drop activations (set to zero) with probability (p)
- During inference: multiply weights by (1 - p) to keep the same distribution (as training)

15. Weight decay

$$E(W; y, t) = \underbrace{E(W; y, t)}_{\text{cross entropy}} + \frac{\alpha}{2} \|W\|_2^2 \quad \xrightarrow{W_1^2 + W_2^2 + \dots} \quad \frac{\partial E}{\partial W} = \frac{\partial E}{\partial W} + \alpha W$$

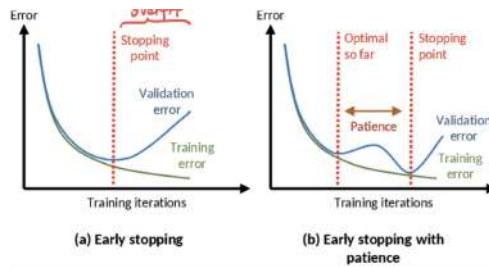
$$W_{t+1} = W_t - \gamma \left(\alpha W_t + \frac{\partial E}{\partial W} \right)$$

- **Reason for decay:**
 - Lowering variance: Prevents the weight from growing too much
 - Keep the model from overfitting
- **Characteristics:**
 - Weight reduction is multiplicative and proportion to the scale of W

16. Early Stopping with Patience

- **steps**
 - In each training iteration observe the validation loss
 - As soon as validation loss starts to increase, start a counter
 - If the validation loss decreases, reset the counter

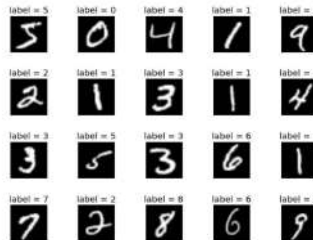
- Otherwise, wait for fixed iterations (patience) and then stop the training



PyTorch Implementation

17. MNIST Dataset

- Input: 28x28 pixel image
- Output: Whether the digit is small (0, 1, 2)
 - output=1 means that the digit is small
 - output=0 means that the digit is not small
- Q: Is this a supervised or unsupervised learning problem?
- Q: Is this a regression or classification problem?



18. ANN

- ANN setup
 - Import all the necessary modules

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
import matplotlib.pyplot as plt # for plotting
import torch.optim as optim
```

```
torch.manual_seed(1) # set the random seed
```

set the seed to some fix num
: will not change when reproducing results

Important for **reproducing** results

- ANN Architecture

PyTorch: ANN Architecture

[1] map the original 28x28 matrix to a vector
(get flattened: 28x28 by 1x1)

Define our neural network architecture

```
# define a 2-layer neural network
class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 30)
        self.layer2 = nn.Linear(30, 1)
```

First layer being input layer
28x28 map to 30

A hyper parameter (put it by chance)
second layer being 30 map to 1

class nn.Linear defines a fully-connected layer

forward pass

```
def forward(self, img):
    [1] flattened = img.view(-1, 28 * 28)
    [2] activation1 = self.layer1(flattened)
    [3] activation1 = F.relu(activation1)
    [4] activation2 = self.layer2(activation1)
    return activation2
```

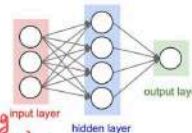
forward() method defines how to make a prediction

[5] Instantiate

```
[5] model = NN()
```

[2] layer 1 → linear func
[3] Pass layer 1 to relu (non-linear)

[4] Can apply a sigmoid activation here but usually don't. since the loss func inside the sigmoid where is the final sigmoid activation?



19. Loss Function and Final Activation for ANN

- You would expect to see the sigmoid activation applied to the output layer.
Indeed this would be the case if we used:

```
criterion = nn.BCELoss()
```

Binary cross entropy loss

- Due to numerical stability, we will use:

```
criterion = nn.BCEWithLogitsLoss()
```

- Applies sigmoid activation internally!
Assume take the raw input as the logits; it implements the sigmoid func inside to avoid overfit

20. PyTorch load data example

Load MNIST data:

```
# load commonly used MNIST dataset
mnist_data = datasets.MNIST('data', train=True,
                             download=True)
mnist_data = list(mnist_data)
mnist_train = mnist_data[:1000]
mnist_val = mnist_data[1000:2000]
img_to_tensor = transforms.ToTensor()
```

- PyTorch also has a number of data loaders to help load new data

21. Forward and Backward Pass

- Forward pass: Make a prediction
 - e.g. `model(input)`, which calls `network.forward` method
 - Information flows forward from input to output layer
- Backward pass: computes gradients for making changes to weights
 - e.g. `loss.backward()`
 - Information flows backward from output to input layer
- Pytorch implementation
 - Training code for binary classification problem

```
#define loss function and optimizer settings
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(pigeon.parameters(), lr=0.005, momentum=0.9)

for (image, label) in mnist_train:
    # ground truth: is the digit less than 3? Create ground truth label
    actual = torch.tensor(label < 3).reshape([1,1]).type(torch.FloatTensor)
    # the gradient accumulates every step
    model = torch.nn.Lstm(1,1)
    out = pigeon(img_to_tensor(image)) # make prediction
    loss = criterion(out, actual) # calculate loss
    loss.backward() # All the gradient will be # obtain gradients
    optimizer.step() # update the weight computed updates parameters
    optimizer.zero_grad() # using lr & momentum a clean up step - important!

    The gradient from previous steps don't affect the current turn.
```

22. PyTorch: Training and Validation Error

- Assessing model performance by tracking error rate and accuracy

```
[[ ]] Two types of error

# computing the error and accuracy on the training set
error = 0
for (image, label) in mnist_train:
    prob = torch.sigmoid(pigeon(img_to_tensor(image)))
    if (prob < 0.5 and label < 3) or (prob >= 0.5 and label >= 3):
        error += 1
print("Training Error Rate:", error/len(mnist_train))
print("Training Accuracy:", 1 - error/len(mnist_train))
```

We set a threshold at `prob = 0.5`

Replace `mnist_train` with `mnist_val` to obtain error and accuracy on test (validation) set

23. Multi-Class Classification

- Requires one-hot encoding at first
- Requires minor changes to our PyTorch implementation:
 - The final output layers has as many neurons as classes
 - Apply the softmax activation function on the final layer to obtain class probabilities

- Use the multiclass cross-entropy loss function
- ANN Architecture multi-class
 - Input size still 28*28, output size cannot be a single neuron

```
class MNISTClassifier(nn.Module):
    def __init__(self):
        super(MNISTClassifier, self).__init__()
        self.layer1 = nn.Linear(28 * 28, 50)
        self.layer2 = nn.Linear(50, 20)
        self.layer3 = nn.Linear(20, 10)

    def forward(self, img):
        flattened = img.view(-1, 28 * 28)
        activation1 = F.relu(self.layer1(flattened))
        activation2 = F.relu(self.layer2(activation1))
        output = self.layer3(activation2)
        return output

model = MNISTClassifier()
```

one output neuron for each of the 10 digits

where is the softmax activation?

24. LossFunction and Softmax Activation

- You would expect to see the **softmax activation** applied to the output layer. Indeed this would be the case if we used:

```
criterion = nn.NLLLoss()
```

negative log likelihood

- Due to **numerical stability**, we will use:

```
criterion = nn.CrossEntropyLoss()
```

- Applies softmax activation internally!

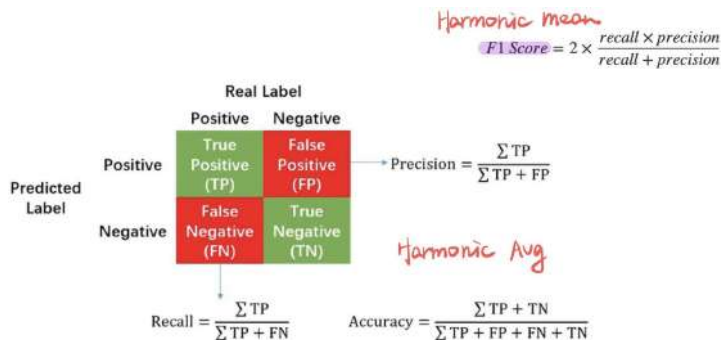
25. Output Probabilities

```
prob = F.softmax(output, dim=1)
print(prob) 10 numbs for each class
print(sum(prob[0]))
```

Evaluating and Debugging

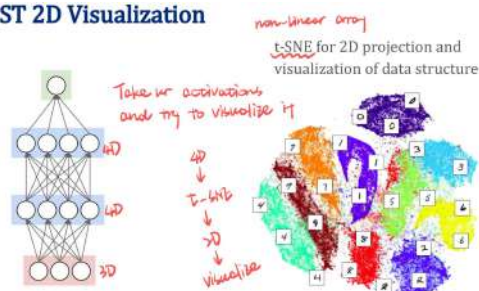
26. Confusion matrix

- What your prediction is with respect to all possibilities



27. MNIST 2D Visualization

MNIST 2D Visualization



28. Debugging NN

- Make sure your model can overfit
 - Make sure you can get loss to decrease w.r.t training data
- Make sure that your network is training: i.e. loss is going down.
 - Sanity check!
- Ensures that you are using the right variable names, and **rule out other programming bugs** that are difficult to discern from architecture issues.
- Confusion Matrix
 - True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN)
- 2D Projections of Data
 - PCA, t-SNE

Introduction to Artificial Intelligence

1. AI

- Reason for developing: reproduce human intelligence with machines
- Captures the notion of developing computer systems that can perform tasks normally only human could
- Statistics -> ML -> AI
- Symbolic Approach:
 - Dominated the early days of AI
 - Like the knowledge of an adult
 - Construct well-defined discrete symbols
 - Too abstract to generalize to real-world
 - Input: data and program, output: result
- Connectionist Approach
 - Dominated AI since 2012
 - Simulate how a baby learns
 - Input: data and result, output: program
 - Requires large-scale data and compute
- Fields of Deep Learning
 - ML
 - Computer vision
 - NLP

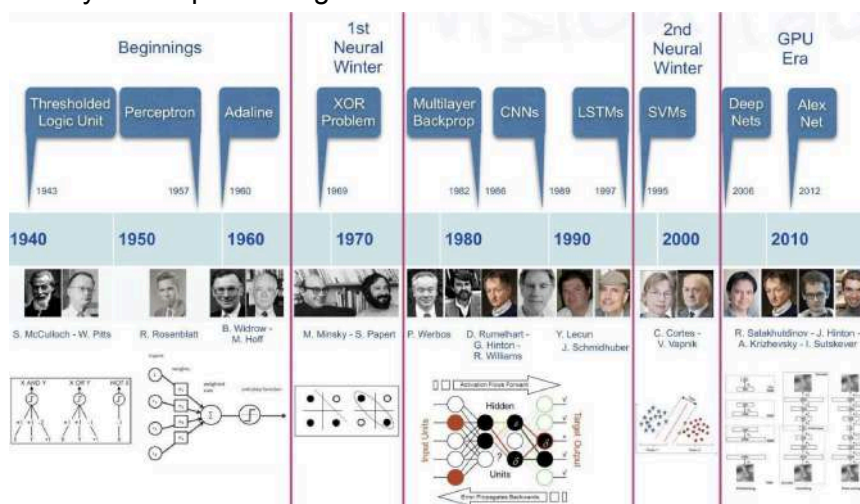
2. Machine Learning

- Formal Definition (Mitchell et al. 1997):
 - Learn from experience (E) with respect to some class of tasks (T) and performance measure (P)
 - Performance at tasks in (T), as measured by (P), improves with experience (E)
- It enables computers to learn from data, avoid tons of hard coding
- Reason for need:
 - Human-generated results will not encounter some counter-example, difficult-to-formulate rules that cover all the conditions
 - We need high-dimensional input space, hard to understand, and must first learn easier representations

3. Deep Learning

- Latest version of ANN, or connectionism an old ML method
- Formal definition (LeCun et al. 2015):
 - A subset of ML
 - Allows multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input), into a representation at a higher, slightly more abstract level.

4. History of Deep Learning



5. Terminology Summary

Artificial Intelligence (AI): broad & poorly defined concept of developing computer systems that can perform tasks normally only humans could do

Machine Learning (ML): computers learn by example, from data, rather than being explicitly programmed, to solve a task

Deep Learning (DL): A machine learning method that learns multiple levels of abstractions over data end-to-end

6. Deep Learning applications

- Machine Translation
- Drug Discovery
- Speech Recognition
- Image Generation
- Alpha Fold
- AlphaGo
- Mathematics
- Code Generation
- Language Modelling
- Simulators

7. Deep Learning Caveats

- Interpretability
- Adversarial examples
- Causality
 - Causality: Relationship between cause and effect (A导致B)
 - Correlation: Association between two variables (A和B有关联, 但A不是导致B的原因)
- Fairness & Bias

8. Bias

- Problems in the collection of training data:
 - A binary classification problem where most of the data comes from one class
 - Data not representing the population

9. Machine Learning Basis

- Supervised Learning
 - Regression (real-valued or continuous value) or Classification (categorical or 1 of N)
 - Requires data with ground-truth labels/outputs
- Unsupervised Learning
 - Self-supervised learning, Semi-supervised learning
 - Requires observations without human annotations
- Reinforcement Learning
 - Sparse rewards from the environment
 - Actions affect the environment (dynamic nature)

10. Supervised Learning

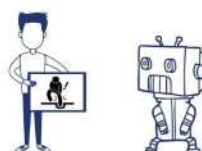
Supervised Learning

Model **learns to map an input to an output based on example input-output pairs.**

Much like a **teacher guides a student**, but with **many** more examples

Examples:

- Age prediction given a headshot:
 - Input: headshot image
 - Output: person's age
- Sentiment classification given a tweet:
 - Input: tweet text
 - Output: whether the tweet is happy or sad



(\mathbf{X}, y)

11. Inductive bias (learning bias):

- the set of assumptions that used for modeling.

12. Mean Squared Error (MSE):

- measures how close a regression line is to a set of data points

13. Error and loss

- Need a way to quantify model performance
 - Minimum error from our fit
- Optimize a metric
 - Proxy called a loss

14. Bias versus Variance Tradeoff

- Greater model complexity higher variance and chance of over-fitting
 - Over-fitting过拟合: occurs when a statistical model fits exactly against its training data.
- Lower model complexity leads to higher bias and under-fitting
 - Under-fitting欠拟合: A data model is unable to capture the relationship between the input and output variables accurately, generating a high error rate on both set and unseen data.

15. Training and Testing Data

- More data -> better model
- If testing data is the same as training data -> over-fitting

16. Validation and Holdout Data

- Split data into train, val, test
- Train on training, tune hyper-parameters on validation, evaluate sparingly on test set (holdout data)
 - Tune: the process of tuning the parameters present as the tuples while we build ML models.