APS360: Applied Fundamentals of Deep Learning

Arnav Patil

University of Toronto

Contents

1 Introduction

1.1 Terminology

- AI – broad and poorly defined concept
- ML – computers learn by example rather than being programmed explicitly
- DL – ML method that learns multiple levels of abstractions end to end

1.2 Deep Learning

- DL is the latest version of artificial neural networks or connectivism, an old ML method
- Neural networks were inspired by the brain
- Concept of 'latent space'

1.3 Caveats of Deep Learning

- Interpretability – can't change inputs till outputs make sense
- Adversarial attacks – e.g. of adding noise to images to classify wrong
- Causality – don't want NNs to just memorize patterns, they should understand context and causality as well
- Fairness and Biases – understanding implicit bias within NNs/models

1.4 Loss Function

- Want to learn coefficients so predictions fall close to the training data, we square everything to keep it positive

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \vec{w}) - t_n\}^2$$

1.5 Avoiding Overfitting

- Regularization – larger weights lead to overfitting, lambda factor determines how much we care for fit vs weight size

$$\tilde{E}(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \vec{w}) - t_n\}^2 + \frac{\lambda}{2}||\vec{w}||^2$$

1.6 Simplified Biological Neuron

- Dendrites – receive information from other neurons
- Cell body – consolidates information from the dendrites
- Axon – passes information to other neurons
- Synapse – area where axon of one neuron to dendrite of another

1.7 Sigmoid

- Most common before 2012, easily differentiable and smooth
  - Hyperbolic tangent
$$f(x) = \tanh(x)$$
  - Logistic function
$$f(x) = \frac{1}{1 + e^{-x}}$$
- Problem – gradients become vanishingly small quickly after 0, bad so we need the chain rule

1.8 ReLU Function

- Modern DL uses rectified linear units
- $\text{ReLU}(x) = \max(0, x)$ – still not smooth or differentiable
- Continuous approximations of ReLU include SiLU and SoftPlus

2 Artificial Neural Networks

2.1 Training Neural Networks

- Make a prediction for some input data x with a known correct output t
- Compare correct output w/ predicted output to compute loss
- Adjust the weights and bias terms to make prediction closer to the ground truth
- Repeat until we have an acceptable loss

2.2 Loss Function

- Want a canonical representation – how to compare predicted label to the true label
- Mean-squared error, cross entropy, binary cross entropy

$$\begin{aligned} \text{MSE} &= \frac{1}{N}\sum_{n=1}^{N}(y - t_n)^2 \\ \text{CE} &= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k} \cdot \log(y_{n,k}) \end{aligned}$$

2.3 Gradient Descent

- Need to know how to change each of our neurons' weights to reduce error
- Find $\partial E/\partial w_{i,j}$ – simple to calculate adjacent to output layer
- Direction of gradient is the direction in which the function increases most quickly, so we have to step the other way

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

2.4 Neural Network Architecture

- Having single decision boundary is not enough most times, XOR needs two decision boundaries for example
- Solve XOR by having at least one hidden neural network layer
- In limit of ∞ wide NN with at least 1 hidden layer, NN is a universal function approximator
- Credit assignment problem – how to train NN of more than 1 layer – just apply more chain rule
- NNs can be viewed as a way of leaving features directly, use activations of layer before as inputs

2.5 Optimizers

- Stochastic gradient descent
  - For each iteration, evaluate a training sample from training data at random
  - Computing gradient takes less but may not actually be faster
  - SGD allows for more global search for optimum, often results in better set of weights
- Mini-batch gradient descent

  - Can apply batching instead of working with one sample
  - Use our network to make predictions for n samples
  - Compute average loss for n samples and take step to optimize average loss
- Ineffective batch size
  - Too small – noisy and optimize different function loss each iteration
  - Too big – expensive average loss might not change very much as batch size grows. True gradient might not always be the best to optimize for
- SGD with momentum
  - Ravines are areas where surface curves more in one direction than another, common around local minima
  - Momentum term increases for dimensions whose gradients point in the direction and decrease vice versa

$$v_{ji}^{t} = \lambda v_{ji}^{t-1} - \gamma \frac{\partial t}{\partial w_{ji}}$$

- Adaptive moment estimation (Adam) – each weight has its own rate

2.6 Normalization

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

There is also exist dropout and weight decay methods, as well as early stopping with patience.

3 Convolutional Neural Networks

- Fairly state-of-the-art but take a lot of data to train – now we use transform vision networks
- Downside to fully-connected networks
  - Computational complexity grows
  - Bad inductive bias – ignores inherent geometry
  - Not flexible – different dimensions require different models

3.1 Convolution Operator

- Function that describes how shape of one function modifies
- Convolution of image I with kernel K
  - Multiply each value in I in range of kernel by corresponding element of kernel K
  - Sum all products and write to new 2D array
  - Slide kernel across all image areas until the whole image is covered

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

3.2 Padding and Stride

- Add 0s around the border of the image before convolving
- Keep width/height consistent with the previous layer
- Keep information around the border of the image
- Distance between two consecutive positions of the kernel is called the stride, allows us to control the output resolution

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

3.3 Pooling Operations

- In fully connected NNs we reduced the number os units before the final layer
- Do this to consolidate information and remove noise
- Strided convolutions, max pooling, average pooling
- Max pooling – pooled layers provide invariance to small translations of the input

3.4 Transfer Learning

- Draw a line between learned embeddings and the classifier, we can swap out as we wish
- Learn general features using a pre-trained feature extractor
- Freeze weights in convolutional network and train only FC layer weights

4 Autoencoders

4.1 Motivation

- Unsupervised learning is where we only have input layers
- Requires large amount of labelled data, obtaining it is expensive
- Find efficient representation of input data that could be used to reconstruct the original input using 2 components
- Encoder – converts inputs into an internal representation, dimensionality reduction
- Decoder – converts internal representation to output, generative network
- Hourglass shape creating a bottleneck layer, lower dimensional representation
- Forced to learn most important features in input

4.2 Stacked Autoencoders

- Autoencoders can have multiple layers: stacked (deep) autoencoders, typically symmetrical with regards to central coding layer
- One way to ensure proper training is to compare the inputs and outputs
- Perfect reconstruction would mean no overfitting

4.3 Denoising Autoencoders

- Noise can be added to input to force model to learn
  - Autoencoders then trained to find original, no-noise inputs
  - Typically we add Gaussian noise
- Applications – feature extraction
  - Unsupervised pre-training
  - Dimensionality reduction
  - Anomaly reduction
  - Generate new data

4.4 Variational Autoencoders

- Probabilistic – their outputs are partly determined by chance even after training
- Generative – they can generate new instances that look like they were sampled from the dataset
- Want encoder distribution to be close to normal as possible – zero mean and unit variance
- Can use Kullback-Leibler divergence to measure the difference between $P(X)$ and $Q(X)$

4.5 Convolutional Autoencoders

- Encoder – learns visual embedding with convolutional layers
- Decoder – up-samples the learned visual encoding to match the original size
- Going from $k \times k$ to 1 was original, now we want 1×1 to $K \times K$
- Padding now has the exact opposite effect we had earlier

5 Recurrent Neural Networks

5.1 Word Embeddings

- Convert words/numbers into symbolic representation
- Autoencoders – learn an embedding space
- How can we learn embeddings of words
- Each word has its own index – input to the encoding
- Decoder – embedding – ??? what is the target
- Meaning of a word is not representation by letters of word
- Meaning comes from the context

5.2 Text as Sequences

- Idea: meaning of word comes from context – think how children learn new words
- Family of architectures used to learn word embeddings
- Skipgram – predict context from the target
- CBOW – predict target from the context

5.3 SkipGram Model

- Given a word predict its neibouring word
- Neighbouring words are defined by the window size – a hyperparameter
- The output layer is only used for training
- After the model is trained, we only keep the weights from input to the hidden layer
- Words that have similar context words will be mapped to similar embeddings
- Characteristics:
  - Works well with small datasets
  - Better semantic relationships
  - Better representation of less frequent words

5.4 Continuous Bag of Words (CBOW) Model

- Predicts the centre word from a fixed window size of context words
- Note that similar to SkipGram the input and output are one-hot representation of a pair of words
  - Train faster than Skip-Gram as the task is simpler
  - Better syntactic relationships
  - Better representation of more frequent words

5.5 GloVe

- Word2Vec does not have any explicit global information, but GloVe enforces global information into the embeddings

5.6 Distance Measures

- In order to talk about which words have similar embeddings, we need to introduce a measure of distance in the embedding space
- Euclidean distance

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

- Cosine similarity

$$\text{Sim}(X, Y) = \cos(\theta) = \frac{X \cdot Y}{||X|| \cdot ||Y||}$$

5.7 Limitations

- Two sentences will have the same embeddings in our model but they can have drastically different meanings
- Our model does not take into account the order of words, so how can we do better?
- Idea 1 – concatenate the word embeddings, then train a neural network that takes the concatenated embedding as input
- Idea 2 – concatenate the word embeddings, then train a 1-dimensional convolutional NN that takes the concatenated embedding as input
- Idea 3 is the Recurrent Neural Networks
  - Can take in variable-sized sequential input
  - Can remember things over time, or has some sort of memory or state
  - Start with an initial hidden state with a blank slate

5.8 Hidden State

- Hidden state is updated based on previous hidden state and the input
- Continue updating the hidden state until we run out of tokens
- Use the last hidden state as input to a prediction network

5.9 Sequence Learning

- In an image, we do not want to learn different weights for every pixel
  - CNNs use convolutional filters with parameter sharing
  - CNNs resue convolutional filters for every pixel
- In a sequence, we do not want to learn different weights for every token
  - RNNs use a shared neural network to update the 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)
- If we consider the concatenated input/hidden and output/hidden vectors as simply input/output, then the forward path in an RNN is simply a fully-connected NN

5.10 Limitations of Vanilla RNNs

- What happens to RNNs unrolled onto a long sequence – two related problems
  - Not good at modelling long-term dependencies
  - Hard to train due to vanishing/exploding gradients
- Tackling exploding and vanishing gradients
  - Gradient clipping for exploding gradients – if greater than a threshold, then clip it
  - Skip connections for vanishing gradients, ideally we want skip connections to all previous states but that's too expensive – we could preserve the hidden state/context over the long term

5.11 LSTMs and GRUs

- We can approximate skip connections to all previous states by learning to weigh previous states differently instead
- Use gates that learn to update the context selectively
- Gating mechanism controls how much information flows through

5.11.1 Long Short-Term Memory (LSTM)

- Consist of a long-term memory (cell state) and a short term memory (context or hidden state)
  - Forget gate – how much of the past memory to forget?
  - Input gate – how much of the current input should contribute to the memory?
  - Updated long-term memory – amount of past that is remembered combined with the memory that was just created
  - Output gate – how much of the updated long term memory should construct the short term memory

5.11.2 Gated Recurrent Unit (GRU)

- GRUs are more efficient while having a similar performance
- Main difference is that they combine the forget and input gates into an update gate
- They also merge cell state into an update state

5.11.3 LSTMs & GRUs vs. RNNs

- LSTMs/GRUs can be trained on longer sequences and are better at learning long-term relationships
- Easier to train and achieve better performance

5.12 Deep and Bidirectional RNNs

- Typical state in an RNN relies on past and present
- In tasks such as machine translation, where a prediction depends on all past, present, and fiture, we can exploit the future to improve performance
- Stack RNN layers to learn more and more abstract representations
- Representations in first layers are better for syntactic tasks while representations in last layers

5.13 Sequence-to-Sequence Models

- Learning to generate new sequences addressing some problems
  - How do we generate variable-length sentences? How do we know when to start/stop
  - Training time behaviour must be changes
  - Inference-time behaviour also changes
- We can use Beginning of Sequence (BoS) and End of Sequence (EoS) characters and learn their embeddings as well
- Teacher forcing – compute the loss by comparing ground-truth and predicted tokens in each step
  - To make training efficient, we force RNN to stay close to the ground-truth sequence
  - Do this by passing ground-truth label as the next input instead of current prediction
- During inference, use greedy search and beam search to help compute next token out of many possibilities

6 Generative Adversarial Networks

6.1 Generative Learning

- Is an unsupervised learning task
  - there is a loss function – an auxiliary task that we know the answer to
  - there is no ground truth wrt to the actual task that we want to accomplish
  - we are learning the structure and distribution of data, rather than labels for data
- unconditional generative models take random noise of a fixed token as an input, with no control over what category they generate
- conditional generative models take one-hot encoding of the target category + random noise or an embedding generated by another model

6.2 Generative Adversarial Networks

- Generator model – try to fool the discriminator by generating real-looking images
- Discriminator model – try to distinguish between real and fake images
- Loss function of the generator is defined by the discriminator

6.3 Loss Function for MinMax Game

- Learn discriminator weights to maximize the probability that it labels a real image as real and a generated image as fake
- Use BCE loss function
- Learn generator weights to maximize the probability that the discriminator labels a generated image as real
- Use discriminator as a loss function

6.4 Problems with Training GANs

6.4.1 Vanishing Gradients

- If the discriminator is too good, then the generator will not learn, remember we are using the discriminator as a loss function for the generator
- If the discriminator is too good, small changes in the generator weights won't change the discriminator outputs

6.4.2 Mode Collapse

- We want the generator to generate a variety of outputs
- If the generator starts producing the same output or set of outputs then the best strategy is for the discriminator to just reject all of those outputs
- But if the discriminator is trapped in local optimum, it cannot adapt to generator and the generator can fool it by only generating one type of data

6.4.3 Failing to Converge

- Due to MinMax optimization process, training Vanilla GANs is difficult
- Difficult to see if there even is progress
- To train faster we use
  - LeakyReLU activations instead of ReLU
  - Batch normalization
  - Regularizing discriminator weights and adding noise to discriminator inputs

7 Transformers

- Vanilla RNNs, LSTMs, and GRUs are sequential in nature
- They are inefficient as we cannot take full advantage of vectorization and parallel GPUs
- When humans read or look:
  - They focus (attend) on some regions within the input – high resolution
  - They pay less attention to surrounding and perceive it less – low resolution

7.1 Attention

- We can use a fully-connected network that takes in word embeddings and generates a single score for each embedding
- We can then normalize these scores across all words within the tweet using a softmax

$$c_i = \sum_j a_{ij} h_j$$

- This network is then trained end-to-end with the classifier

7.2 Attention in Transformers

- Transformers are a class of deep models that are based on self-attention where attention is modelled as a neural dictionary
  - It retrieves value v_i for a query q based on a key k_i
  - Values, queries, and keys are d-dimensional embeddings
  - However, rather than retrieving a single value for a query it uses a soft-retrieval
  - Retrieves all values but then computes their important with regards to the query based on the similarity between the query and their keys

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

- Suppose X is an input sequence consisting of n tokens where each token $t \in \mathbf{R}^i$
- Self-attention in transformers is defines as follows:

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

7.2.1 Multi-Head Attention

- To improve the performance we can take three steps:
- Divide the representation space to h subspaces
- Run parallel linear layers and attentions
- Concatenate them back to form the original space

7.2.2 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

$$\begin{aligned}\text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\ FFN(x) &= \max(0, xW_1 + b_1)W_2 + b_2 \\ &\text{LayerNorm}(x + \text{Sublayer}(x))\end{aligned}$$

7.2.3 Positional Encoding

- The model does not have recurrent or convolutional layers so it does not take into account the order of sequence
- We use positional encoding to make use of the order which allows the model to easily learn to attend by relative position

In short, we can compare RNNs and Transformers as follows:

- RNN:
  - Struggling with long range dependencies
  - Gradient vanishing and explosion
  - Large number of training steps
  - Recurrent prevents parallel computation
- Transformer
  - Facilitate long rage dependencies
  - Less likely to have gradient vanishing and explosion problem
  - Fewer training steps
  - No recurrent, facilitates parallel computation

7.3 Transformers for Language Modelling

- We can use a self-supervised objective such as predicting the next work to learn embeddings over tokens
- Word2Vec/Glove learns static embeddings with one embedding for all senses
- RNNs/Transformers learn contextual embeddings, embeddings of a same word changes according to the sentence it appears in
- Transformer models differ in terms of their prediction tasks and training objectives

7.3.1 Task 1: Masked Word Prediction

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

7.3.2 Task 2: Next Sentence Prediction

- Given two sentences, predict if they appear together
- Create 50% positive and 50% negative pairs of sentences

7.4 Transformers for Computer Vision

- Vision transformers (ViT) are starting to dominate computer vision
- Compared to CNNs they achieve higher accuracies on large datasets due to their
  - Higher modelling capacity
  - Lower inductive biases
  - Global receptive fields
- CNNs are still on-par or better than ViTs on ImageNet in terms of model capacity or size versus capacity

8 Graph Neural Networks

8.1 Graphs

- Graph $G = (V, E, X)$ is a data structure that encodes pair-wise interactions or relations among concepts and objects
- V is a set of notes 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 the number of edges connecting to that node

8.2 Transformers and Graphs

- What happens if we omit the positional encoding from transformers?
  - The transformer learns an $N \times N$ attention matrix which represents pairwise importance scores
  - This means transformers create a fully-connected graph over the input and learns the edge weights

8.3 Message-Passing

- GNNs learn embeddings over graphs through message passing
- For each node in graph:
  - Aggregate embeddings of its neighbour nodes
  - Combine the aggregated embedding with the node embedding
  - Update the node embedding
- Aggregate function must be an order invariant function

$$h_v^{(k)} = \text{COMBINE}\big(h_v^{(k-1)}, \text{AGGREGATE}\big(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\}\big)\big)$$

8.4 What Can GNNs Do?

- We can use them to predict links between nodes
- Node classification

$$z_i = f(h_i)$$

- Graph classification

$$z_G = f(\oplus_{i \in V} h_i)$$

- Link prediction

$$z_{ij} = f(h_i, h_j, e_{ij})$$

8.5 Graph Convolutional Networks

- A layer of a GNN is a nonlinear function over node features and an adjacency matrix

$$\mathbf{H} = \text{ReLU}(\mathbf{AXW} + b)$$

- Limitation 1 – multiplication with A means that for every node we sum up all the feature vectors of all the neighbouring nodes but not the node itself
  - Fix is to add self-loops (add the identity matrix to A)

$$\mathbf{A} = \mathbf{A} + \mathbf{I}$$

- Limitation 2 – A is not normalized and therefore the multiplication with A will completely change the scale of the feature vectors
  - Fix is to symmetrically normalize A using diagonal degree matrix D such that all rows sum to one

$$\mathbf{A} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$$

- Now we can define a GCN layer as:

$$\mathbf{H} = \text{ReLU}(\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}\mathbf{X}\mathbf{W} + b)$$

8.6 Going Deeper With GNNs

- A GCN layer updates the node embeddings based on the features of the immediate neighbours
- We can influence the embeddings from further neighbourhood by stacking GCN layers
- This is analogous to increasing the receptive field in CNNs

$$\begin{aligned}
\mathbf{H}^{(0)} &= \mathbf{X} \\
\mathbf{H}^{(1)} &= \text{ReLU}\big(\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}\mathbf{H}^{(0)}\mathbf{W}^{(1)} + b^{(1)}\big) \\
\mathbf{H}^{(2)} &= \text{ReLU}\big(\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}\mathbf{H}^{(1)}\mathbf{W}^{(2)} + b^{(1)}\big) \\
&\cdots \\
\mathbf{H}^{(l)} &= \text{ReLU}\big(\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}\mathbf{H}^{(l-1)}\mathbf{W}^{(l)} + b^{(l)}\big)
\end{aligned}$$

8.7 Graph Attention Networks (GATs)

- Idea: Instead of using node degree, learn an attention score between two nodes, that is, learn the contribution weight of neighbour nodes
  - Use a shared neural network to compute an attention score betwen two nodes

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

  - Normalize the attention scores

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

  - Update the node embeddings based on the attention score

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

8.8 Implementation

- Most graphs are sparse
- Dense implementation uses N^2 space for adjacency
- Implementing sparse operations are supported by Python but not very straightforward – we can use PyTorch Geometric (PYG)
- Dense implementation – batching is done by creating a diagonal matrix of adjacency matrices
- Sparse implementation – uses an index vector which maps each node to its respective graph in the batch