A2 DEEP LEARNING

1. GENERATIVE ADVERSARIAL NETSWORKS

Normally, one has $x \sim p_{\text{data}}$ to learn a p_{model} and generate $y \sim p_{\text{model}}$. If p_{model} is a high dimensional complex distribution, there is no direct way to generate new samples. Instead, GANs try to learn a transformation which can go from $x \sim p_{\text{data}}$ to $y \sim p_{\text{model}}$. It samples from a simple distribution and learns a transformation to a sample of a target distribution.

Training GANs is a two-player game, where you have a generator network, which will try to fool the discriminator by generating realistic images, and a discriminator network, which tries to distinguish between real and fake images. Minmax objective function:

$$\min_{\theta_g} \max_{\theta_d} [\mathbb{E}_{x \sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p_{(z)}} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))]$$

Note: $\log D_{\theta_d}(x)$ represents the disciminator output for real data x, while $D_{\theta_d}(G_{\theta_g}(z))$ is its output for fake data.

(1) Both are initialized with random weights; (2) for each iteration, generator outputs an image from a random seed; (3) discriminator classifies the example as real or fake; (4) generator aims to minimize the loss, which represents how well it is fooling the discriminator (neg-log-lik of discriminator being correct). Discriminator loss reflects how well it is distinguishing between real and generated samples.

In other words, discriminator wants to maximize objective such that $\log D_{\theta_d}(x)$ is close to 1 for real and $D_{\theta_d}(G_{\theta_g}(z))$ is close to 0 for fake. The generator wants to minimize objective such that $D_{\theta_d}(G_{\theta_g}(z))$ is close to 1 (discriminator is fooled).

Conditional GANs conditions the train and generation steps over some condition x, instead of only conditioning under the noise. In the minmax problem, both generator and discriminator receive another parameter x which stands for the condition.

Cycle GANs is a way to learn how to translate image from a source domain $\mathbf{X}$ to a target domain $\mathbf{Y}$. Zebras into Horses, Paintings into photos etc. We can train G to generate horses and F to generate zebras, both from noise. (Adversarial) Loss functions are $\mathcal{L}_{\text{GAN}}(G, D_{\mathbf{Y}}, \mathbf{X}, \mathbf{Y})$ and $\mathcal{L}_{\text{GAN}}(F, D_{\mathbf{X}}, \mathbf{Y}, \mathbf{X})$. We can calculate the loss for $F(G(x)) \approx x$ and vice-versa with $\mathcal{L}_{\text{cyc}}(G, F)$. Also, we can calculate the identity loss, $\mathcal{L}_{\text{iden}}$, which means that if you feed image y to generator G, it should yield something close to image y. The full loss is $\mathcal{L}(G, F, D_{\mathbf{X}}, D_{\mathbf{Y}})$, which controls the relevance of each loss term through parameters $\lambda_{\text{GAN}}, \lambda_{\text{cyc}}, \lambda_{\text{iden}}$.

2. AUTOENCODERS

It aims at reproducing the input at the output. To do so, it comprises an encoder that contracts the data, followed by a decoder that recovers the original dimension. At the bottleneck, we have a latent space ("true" explanatory variables). The loss relates to the similarity input $\longleftrightarrow$ output: $\mathcal{L}(W) = \frac{1}{N} \sum_i ||x_i - \hat{x_i}||^2$.

The autoencoder features (latent data representation) can be the input to a machine learning model. It is a good way to reach important variables once it can learn non-linear relations.

Convolutional Autoencoders aim at reproducing the input at the output.

Denoising Autoencoders add noise to a clean image to crate a noisy version of it, an reconstruct the output from that noisy input. This aims at learning a compact representation of the input by training the model to remove noise it. As the encoder learns to capture essential features from data, the model becomes more robust to variations and noise.

Variational Autoencoders use aspects of GAN. The generator maps samples z from a known distribution $q(z)$ to a target distribution $p(x)$. But we do not know which z leads to desired output x.

Assume that the distribution of an image x is determined by latent variables z. Some methods allow estimating $p_\theta(x|z)$, while $p_\theta(z|x)$ is intractable. However, we can approximate (KL Divergence) it by another tractable distribution $q_\phi(z|x)$, $\mathcal{N}(\mu, \Sigma)$, so we can sample the z values that better correspond to the given x we want to reconstruct (slide 38).

The VAE loss comprises 2 terms:

$\mathcal{L}(\phi, \theta, x) =$ reconstruction loss + regularization loss. The reconstruction loss is the distance between input and output, while the regularization term is the KL Divergence.

Instead of producing z, the encoder generates μ and Σ and samples from it. But this does not allow backprop to flow through the node. So we use the reparametrization trick to get $z = \mu + \text{diag}(\Sigma) \odot \epsilon$.

3. SIMILARITY LEARNING

One-Shot Learning is about looking at two images never seen before and saying whether they represent the same object. Instead of learning how to classify the input, learn how to **measure similarity** of two inputs.

Learn an embedding space $\mathbb{R}^k$ where images of the same person are close and of different people are far away.

(1) We find a mapping $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^k$ to the embedding space with those properties; (2) choose a distance function related to the (dis)similarity between inputs $x^{(1)}$ and $x^{(2)}$. The distance is large for different people and small for the same people; (3) define a threshold τ to decide whether they are the same of different.

Siamese Networks are two identical networks that share their weights: $d(x^{(1)}, x^{(2)}) = ||f(x^{(1)}) - f(x^{(2)})||^2$. Then we update the weights for each branch independently, and average them.

Siamese Networks – Contrastive Loss: similar (positive) pair should have small distance, while dissimilar (negative) pair must have large distance and set loss to zero if the distance exceeds a margin m^2. So it is $\max(0, m^2 - ||f(x^{(1)}) - f(x^{(2)})||^2)$. The general formula for all pairs is $\mathcal{L} = \sum_i y_i ||f(x_i^{(1)}) - f(x_i^{(2)})||^2 + (1 - y_i)\max(0, m^2 - ||f(x_i^{(1)}) - f(x_i^{(2)})||^2)$, $y_i = 0$ if dissimilar, $y_i = 1$ if similar.

Siamese Networks – WDMC: if similar, distance is small and loss is set to zero if it is lower than a margin m_1^2: $\max(0, ||f(x^{(1)}) - f(x^{(2)})||^2 - m_1^2)$. If dissimilar, exactly the same as for the constrastive loss. The general formula for all pairs is $\mathcal{L} = \sum_i y_i w_i \max(0, ||f(x^{(1)}) - f(x^{(2)})||^2 - m_1^2) + (1 - y_i) w_2 \max(0, m_2^2 - ||f(x_i^{(1)}) - f(x_i^{(2)})||^2$. w_1 and w_2 are weights for similar/dissimilar pairs.

The intuition behind the m terms is to avoid the effort to move closer than m_1 or further than m_2.

Siamese Networks – Triplet Loss: use an arbitrary sample A. We want $||f(A) - f(P)||^2 < ||f(A) - f(N)||^2$, P being a

same class anchor and N being a different class anchor. To make it more restrictive, we can use a loss fuction such as $\mathcal{L}(A, P, N) = \max(0, ||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + m)$.

Formulation considering all triplets: $\mathcal{L} = \sum_i \max(0, ||f(A_i) - f(P_i)||^2 - ||f(A_i) - f(N_i)||^2 + m)$. We train using a 3-branch network to compute each of the two distances from A. They share weights. Testing is the same as the contrastive case.

Improving Similarity Learning: (1) using different loss functions; (2) sampling; (3) Ensembles — multiple networks, each trained with a subset of triplets (divide and conquer, for instance); (4) use the Classification loss for similarity learning.

Change Detection involves finding the changes in images of the same location taken at different times.

4. UNSUPERVISED LEARNING

We can use **DEEP CLUSTERING** with an efficient representation mapping (i.e., using autoencoders + k-Means). It has two phases: (1) parameter initialization with a deep autoencoder; (2) parameter optimization (clustering) where we iterate between computing an auxiliary target distribution and minimizing the KL Divergence to it.

$\mathcal{L} = \sum_x \mathcal{L}_{\text{rec}}(x) + \lambda \mathcal{L}_{\text{clust}}(x)$

$\mathcal{L} = \sum_{x \in \mathcal{X}} g(x, A(x; \theta)) + \lambda \sum_{k=1}^{K} f(h_\theta(x), r_k) G_{k,f}(h_\theta(x), \alpha; \mathcal{R})$

G is a temperature-scaled softmax, which means that higher values of α sets a calibration level at which the softmax is sharper, and the models is more confident of its predictions. With $\alpha \to 0$, each data point in the embedding space is equally close, which corresponds to removing the influence of $G_{k,f}$; $\alpha \to 1$, each data point is proportionally close to cluster centroids considering $G_{k,f}$, corresponding to a soft assignment; $\alpha \to \infty$, each point in the embedding space belongs to a single cluster, corresponding to a hard assignment.

5. SELF ATTENTION

It is a mechanism by which a network can weight feature by level of importance to a task and use this weighting to help achieve the task. NLP context, it looks at the other words at the other words in the setence and weights their influence on the definition of the meaning of the word of interest — **adding contextual information** to the word in the sentence.

For time series, for example, it decreases the influence of far away data and amplifies the influence of close data. But in case of words, relation has little to do with proximity, and much to do with linguistic meaning.

Self Attention for NLP: we have to vectorize words (input embedding) in order to apply "linear combinations" such as $\text{King} - \text{man} + \text{woman} = \text{Queen}$, according to the words properties, which weights are inside the vector that represents the word. It also adds (literally) positional encoding (so that we can also take position into account) using sine and cosine functions.

We use the dot product to measure similarity. The direction of the vector is related to semantics, while the magnitude is related to the semantic strength. If two vectors are large and close to one another, their dot product is going to be big — in other words, you get a big dot product one the two words have strong and similar semantic content.

But how to add more context? Consider the phrase "Bank of the river". It is expected that "bank" is more bound to water than to money. Self attentions allows the model to associate each word with the other inputed words.

To achieve it, we (1) feed the input into 3 distinct fully connected layers to create Query, Key and Value. They are each a separate linear transformation of the same exact input (word embeddings); (2) we apply the dot product between Query and Key to get a score matrix, which determines how much attention should a word put on other words. Heigher the score, higher the focus; (4) We then normalize it by the dimension of the queries and keys, which is import to stabilize the gradients; (5) applying the softmax to this matrix, we have the attention weights; (6) we multiply the attention weights with the value vector to get an output vector with encoded information of how each word should attend to all each words in a sequence.

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

To make this a multi-headed attention, we need to split the Query, Key, and Value into n vectors before applying self attention. Each self attention process is called a head, and the output vector of each one gets concatenated before going through the final linear layer. In theory, each head would learn something different, giving the model more representation power. Using multi-heads is useful to learn different representations simultaneously (in parallel). Then we have W_i^Q, W_i^K, W_i^V, where i indexes over the heads. Those are learnable parameter matrices and will project each of the word representations to different Q, K and V vectors.

$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, ..., head_h)W^0$

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

The **TRANSFORMER** architecture itself contains two blocks: an encoder and a decoder. At the `Add & Norm` block, the Multi-Headed Attention output vector is added to the original input (residual connection). This output goes through a Layer Normalization. Then, it goes through a Linear Layer, a ReLu and another Linear Layer, and further add and normalize. This avoids the vanishing gradient problem.

The decoder's job is to generate text sequences. It receives the target sequence as input, and has similar sublayers as the encoder. The Queries and Keys come from the previous layer of the decoder, but the Values come from the output of the encoder, what makes sense as we are conditioning on the input sequence. Another difference is that the decoder is capped off with a linear layer that acts like a classifier and a softmax to get the word probabilities.

OBS.: Masking, applied in the first multi-headed attention layer, avoids that a word compute scores for words that come after it, positionally speaking. Since the decoder is autoregressive and generates sequences word-by-word, you need to prevent it from conditioning at future words.

The gains of using Transformers instead of RNNs is that while RNNs have short reference window and its computation is strictly sequential, Transformers can be parallelized to fasten training