SAiDL Assignment

Raj D.

July 2024

Contents

1 Introduction

This is my first attempt at any such assignment/project. It was a rather interesting experience; after learning something I felt like I could always dig slightly deeper, or understand it better. Of course, if I had tried to do that, I would have never been able to complete the assignment itself. Yet, nonetheless I have tried digging deep where I thought it mattered.

Since this was something new to me, you will find that I have often improvised or bootstrapped solutions as optimally as I thought I could have, so I am hopeful you will be patient with me when you come across topics that seem to be done in an amateurish way.

Since I had never particularly done AI/ML/DL to a great extent before, I thought it best to first proceed with learning how it works, using the resources provided (and some I found along the way). So I will be first documenting the learning process and then move onto the specific tasks in the later sections. The core ML task focused on evaluating the robustness of various loss functions against label noise in image classification (Section 3), while the second task involved building a pipeline for multimodal Graph Neural Network (GNN) node classification (Section 4).

2 Beginning: The Learning Phase

Here I will document and summarize most of the stuff I learned before directly starting with the tasks.

2.1 CS231n: Convolutional Neural Networks for Visual Recognition

All my prior experience with ML and AI was a very simple multi-perceptron model I had made in Java many years ago based on the Hopfield network, so though I had some very basic idea of how it worked, it wasn't as accurate nor precise as I had previously thought.

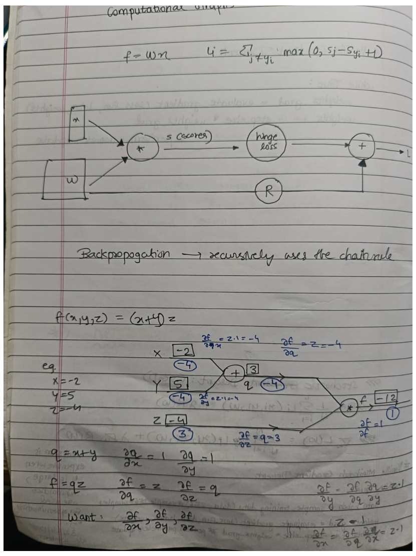
I looked online a bit and it was recommended to me by a member of SAiDL as well that CS231n would probably satisfy my needs and allow me to better understand neural networks to approach this assignment. So that's what I did.

I attempted to follow the entire CS231n course almost like our usual college course, I took notes per lecture, tried to complete the assignment they gave, and tried to understand overall how things work. Unfortunately, I realized midway that I would very likely not be able to complete the course itself as intended on top of my classes + assignment + the rest of the general responsibilities. Frankly, their (CS231n's) assignment itself was taking far too much of my time and thus I decided to take what I had learned and just dive into the tasks.

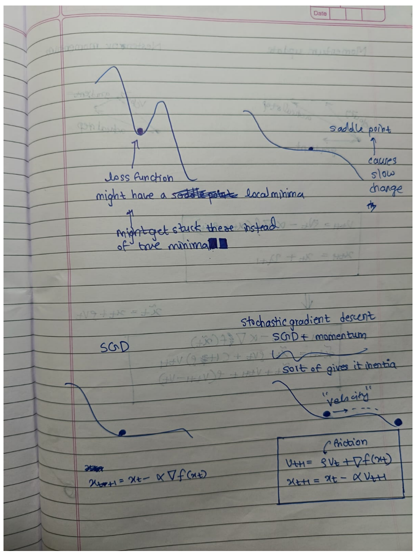
For the next couple of subsections though, I would like to detail everything I managed to learn including the material of CS231n, and give you all a glimpse on how I tried to learn how AI/ML and DL works.

2.1.1 Notes from CS231n

Here in Figure 1 are some examples of notes I took while trying to follow the course. Of course, the number of pages in my notes is much greater than shown here and I will insert all the different notes including ones not from CS231n notes whenever I feel they aid me in illustrating my points better.

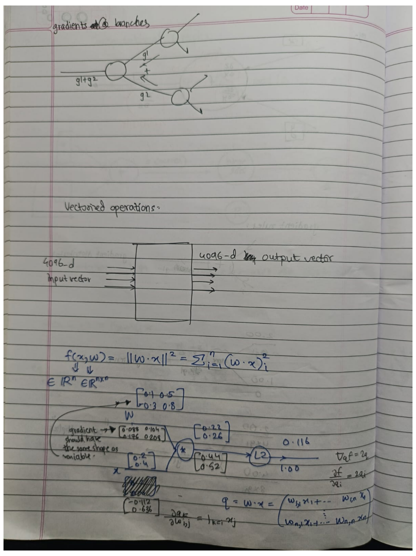


(a) Backpropagation



loss function might have a local minima
might get stuck there instead of true minima
saddle point causes slow change
SGD
stochastic gradient descent
SGD + momentum
sort of gives it inertia
"velocity"
friction

(b) Analogy for momentum application



gradients add at branches
Vectorized operations
4096-d input vector
4096-d output vector
gradient should have the same shape as variable

(c) Backpropagation in vectors simple example

Figure 1: Some random examples to illustrate my attempt at going through CS231n.

2.2 Concepts and Topics in Deep Learning

These are not exhaustive nor all the topics that I have covered; they are simply the ones that I used to get my understanding of ML/DL up to speed. Some of them I looked up only due to interest and may not be completely relevant, and the list itself is from my notes.

2.2.1 Vectorization (DL)

This topic was visited to understand why we use vectors instead of for loops in python, when both would technically work.

- **Non-Vectorized**: You would require the use of 'for loops' in the neural network. This takes significant time, especially since Python's 'for loops' can be slow for numerical computations compared to optimized libraries.
- **Vectorized**: Fast, efficient. It leverages optimized low-level libraries (like BLAS, MKL) often implemented in C or Fortran. These libraries can utilize:
  - *Parallelism:* Multiple CPU cores or GPU cores perform many simple calculations simultaneously (SIMD - Single Instruction, Multiple Data).
  - *Optimized Code:* Highly optimized C/Fortran code is much faster than interpreted Python loops.
  - *Memory Locality:* Libraries like NumPy store arrays of the same data type contiguously in memory. This allows for faster access due to CPU caching (compared to Python lists which can store objects anywhere in memory).

2.2.2 Stochastic Gradient Descent (SGD)

Essentially, in the usual (*batch*) gradient descent, the gradient of the loss function is calculated using the *entire* training dataset for each parameter update. In stochastic gradient descent (SGD), the gradient is estimated using only a *single* randomly chosen training example per update. A common compromise is *mini-batch* SGD (see 2.2.13), which uses a small batch of examples.

SGD is computationally cheaper per update than batch GD. While each update is 'noisier' in a sense, it ends up somehow outperforming Vanilla GD, some possible explantions given are:

- *Escaping Saddle Points and Local Minima:* The noise can help the optimization process "jump out"("buzz out"? that's sort of how I imagine it) of saddle points that it might otherwise get stuck in. I feel like its similar to momentum, not how it works but what we get out of it.
- *Implicit Regularization:* The noise might sometimes be preventing the model from overfitting too tightly to the training data.

Although it might seem slower due to the noise, since it is computationally cheaper, you can simply do it more often, this often leads to faster convergence and sometimes better generalization (finding flatter minima) compared to batch GD(especially for large datasets).

2.2.3 Softmax

Essentially, applying the softmax function, typically on the output of the final fully connected layer in a classification network, allows us to convert the raw output scores (logits) into a probability distribution over the classes. What it does is take a vector of real numbers and transform it into a vector of values between 0 and 1 that sum to 1. Each output value can then be interpreted as the model's estimated probability that the input belongs to that specific class. The formula for an output vector $\mathbf{z}$ is: $softmax(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$ for each element i.

2.2.4 Regularization

Regularization refers to techniques used to prevent overfitting in machine learning models. Overfitting occurs when a model learns the training data too well, including its noise and idiosyncrasies, leading to poor performance on unseen data. Common regularization techniques include:

- *L1 and L2 Regularization (Weight Decay):* Adding a penalty term to the loss function based on the magnitude of the model weights (L1 uses the sum of absolute values, L2 uses the sum of squared values). L2 (weight decay) is very common in deep learning.
- *Dropout:* Randomly setting a fraction of neuron activations to zero during training, forcing the network to learn more robust representations that don't rely on specific neurons.
- *Data Augmentation:* Artificially increasing the size and diversity of the training dataset by applying transformations (like rotation, cropping, flipping for images).
- *Early Stopping:* Monitoring the model's performance on a validation set during training and stopping when performance starts to degrade, preventing the model from overfitting during later epochs.

2.2.5 Activation Functions

Activation functions introduce non-linearity into neural networks, allowing them to learn complex patterns beyond simple linear relationships. They are applied element-wise to the output of a layer (e.g., after the linear transformation in a dense layer or convolutional layer). Common examples include ReLU (2.2.15), Leaky ReLU (2.2.16), Sigmoid, Tanh, and GeLU. Without non-linear activation functions, a deep neural network would simply collapse into an equivalent single linear transformation. *[Further details on specific functions like Sigmoid, Tanh could be added here if needed.]*

2.2.6 Affine Layer (DL)

In the context of deep learning, an "affine layer" or "affine transformation" usually refers to a fully connected layer (also called a dense layer or linear layer) that performs a linear transformation followed by adding a bias term. Mathematically, for an input vector $\mathbf{x}$, the output $\mathbf{y}$ is calculated as $\mathbf{y} = W\mathbf{x} + \mathbf{b}$, where W is the weight matrix and $\mathbf{b}$ is the bias vector. This is a fundamental building block in many neural networks.

Example Figure:

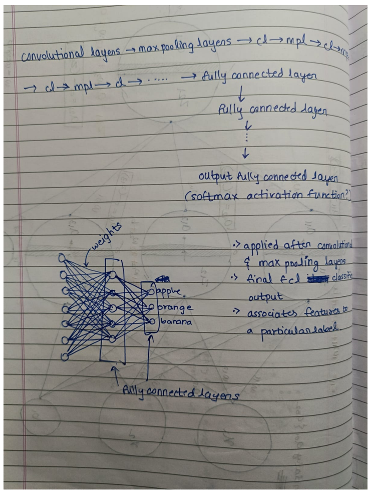

convolutional layers → max pooling layers → cl → mpl → cl → mpl
→ cl → mpl → cl → ..... → fully connected layer
fully connected layer
output fully connected layer (softmax activation function?)
weights
apple
orange
banana
fully connected layers
applied after convolutional & max pooling layers
final fcl classifies output
associates features to a particular label

Figure 2: Diagram illustrating an affine layer (fully connected layer with bias).

2.2.7 Gradient Descent

Gradient descent is one of the main optimization algorithm used to train most neural networks. It iteratively adjusts the model's parameters (weights and biases) to minimize the loss function. In each step, it calculates the gradient (the vector of partial derivatives) of the loss function with respect to each parameter. It then updates the parameters by moving them slightly in the opposite direction of the gradient, scaled by a learning rate. The update rule is typically: $\theta_{new} = \theta_{old} - \alpha \nabla_\theta J(\theta)$, where θ represents the parameters, $J(\theta)$ is the loss function, $\nabla_\theta J(\theta)$ is the gradient, and α is the learning rate. Variations:

- *Vanilla Gradient Descent*
- *Stochastic Gradient Descent*
- *Adaptive Gradient Descent*
- *Momentum Gradient Descent*

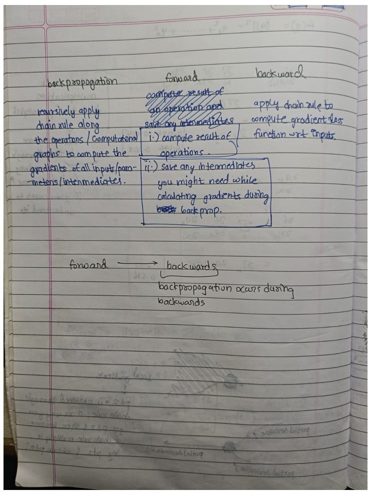
ADAM, RMSprop, AdaGrad, SGD with Momentum are all variants of Gradient Descent, differing in how exactly they use the gradients to perform this update step (e.g., ADAM uses adaptive learning rates and momentum).
ADAM is a very popular algorithm used in place of SGD.

2.2.8 Forward propagation

Input is fed into the model, and layer by layer the data is processed and moves forward till the last layer, where it gives the entire networks prediction.

2.2.9 Backpropagation

Caluclates the gradient of the loss function w.r.t. to bias and weights, starts with the final output and recursively goes back to previous parameters, and thus caluclates the gradient.
Gradient Descent then essentially implements this by applying those gradient changes on the parameters in the opposite direction and thus updating them.



backpropagation

recursively apply chain rule along the operations / Computational graphs to compute the gradients of all inputs/para-meters/intermediates.

forward

~~compute result of an operation and save any intermediates~~

i.) compute result of operations.

ii.) save any intermediates you might need while calculating gradients during ~~back~~ back prop.

backward

apply chain rule to compute gradient loss function wrt inputs.

$$\text{forward} \longrightarrow \underbrace{\text{backwards}}_{\text{backpropogation occurs during backwards}}$$

Figure 3: notes on forward and back prop

2.2.10 Loss Functions

A loss function (or cost function) measures how well the model's predictions match the actual target values. The goal of training is to minimize this function. The choice of loss function depends on the task:

- *Regression:* Mean Squared Error, Mean Absolute Error (MAE).
- *Binary Classification:* Binary Cross-Entropy (Log Loss).
- *Multi-class Classification:* Categorical Cross-Entropy, Sparse Categorical Cross-Entropy.

More advanced or specialized loss functions exist for specific problems, like handling noisy labels (discussed in Section 3.3).

2.2.11 Linear Regression

A basic statistical model used to predict a continuous target variable based on one or more input features, assuming a linear relationship between them. The model learns coefficients (weights) for each feature and an intercept (bias) to define a best-fit line (or hyperplane in higher dimensions) through the data, typically by minimizing the Mean Squared Error (MSE). While simple, it's a foundational concept for understanding more complex models.

2.2.12 Logistic Regression

Despite its name, logistic regression is a model used for *binary classification* problems (predicting one of two outcomes, e.g., yes/no, spam/not spam). It models the probability of the default class using the logistic function (sigmoid function) applied to a linear combination of the input features. The parameters are typically learned by minimizing the Binary Cross-Entropy loss. It forms the basis for understanding classification in neural networks.

2.2.13 Mini-batch Gradient Descent

This is a compromise between Batch Gradient Descent (2.2.7) and Stochastic Gradient Descent (SGD) (2.2). Instead of using the entire dataset (Batch GD) or a single example (SGD) to compute the gradient for each update, Mini-batch GD uses a small, randomly selected subset (a "mini-batch") of the training data (e.g., 32, 64, 128 examples). Advantages:

- More computationally efficient than Batch GD.
- Less noisy gradient estimates than SGD, leading to more stable convergence.
- Allows for parallelization and vectorization (**??**) benefits when processing the mini-batch, making it much faster in practice on modern hardware (CPUs/GPUs) than pure SGD.

It is the most common variant of gradient descent used in deep learning.

2.2.14 Big O Notation in ML/DL

Big O notation is used to describe the limiting behavior (computational complexity or space complexity) of an algorithm as the input size grows. In ML/DL, it helps understand how training time, prediction time, or memory usage scales with factors like:

- Number of training examples (N)
- Number of features (D)
- Number of parameters (e.g., weights in a neural network, W)

- Number of layers (L)

For example, a forward pass in a dense neural network might be roughly $O(L \times W_{avg})$, where W_{avg} is the average number of weights per layer. Training complexity often involves N multiplied by the forward/backward pass complexity per epoch. Understanding this helps in choosing algorithms and anticipating performance bottlenecks.

2.2.15 ReLU (Rectified Linear Unit)

A very popular activation function in deep learning. It's defined as $f(x) = max(0, x)$. Advantages:

- Computationally efficient.
- Avoids the vanishing gradient problem for positive inputs (gradient is 1).(essentially when backpropagation occurs, the values used to update the gradients keep getting smaller and smaller when they are lower than one, as they keep getting multiplied by other <1 gradients, which shrinks them down very rapidly)
- Tends to produce sparser activations than sigmoid or tanh.

Disadvantage:

- "Dying ReLU" problem: Neurons can get stuck in a state where their output is always 0 for any input (if the input to ReLU is always negative), and gradients won't flow through them anymore.

2.2.16 Leaky ReLU

An attempt to fix the "dying ReLU" problem. Instead of outputting 0 for negative inputs, it outputs a small positive slope (e.g., 0.01 times the input). Defined as $f(x) = max(\alpha x, x)$, where α is a small constant (like 0.01). This allows some gradient to flow even for negative inputs.

2.2.17 Momentum

Momentum is an optimization technique often used with SGD or its variants. It helps accelerate gradient descent in relevant directions and dampens oscillations. It introduces a "velocity" term that accumulates a fraction of the past gradients. The update rule becomes: $v_t = \beta v_{t-1} + (1 - \beta)\nabla_\theta J(\theta)$ (sometimes simplified as $v_t = \beta v_{t-1} + \alpha \nabla_\theta J(\theta)$) $\theta_{new} = \theta_{old} - v_t$ (or $\theta_{new} = \theta_{old} - \alpha v_t$ in the second formulation) Here, β is the momentum coefficient (e.g., 0.9). The velocity term v_t acts like a moving average of the gradients, helping the updates to keep moving in a consistent direction and pass through flat regions or shallow local minima faster. It also helps smooth out the noisy updates from SGD/mini-batch GD.
Personally I just like to imagine it as literally like momentum of the updates in a sense.

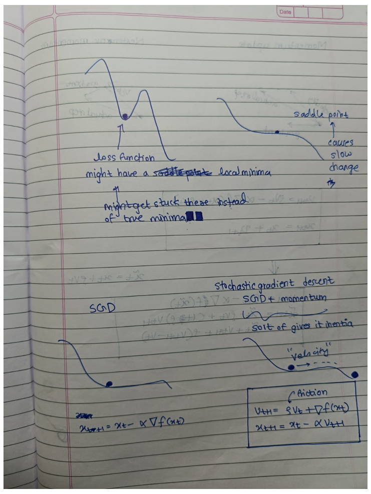

loss function
might have a ~~saddle point~~ local minima
might get stuck these instead of true minima
saddle point
causes slow change
stochastic gradient descent
SGD + momentum
sort of gives it inertia
"velocity"
SGD
friction
$v_{t+1} = \rho v_t + \nabla f(x_t)$
$x_{t+1} = x_t - \alpha v_{t+1}$
$x_{t+1} = x_t - \alpha \nabla f(x_t)$

Figure 4: analogy for momentum

3 Core ML Task: Robustness to Label Noise on CIFAR-10

After gaining some foundational understanding from CS231n and related concepts, I tackled the first core task. The thing is, I realized after doing CS231n that I still didn't really have a complete idea on how the entire CNN thing worked in practice, especially putting all the pieces together in code. So, I tried to get that down first.

3.1 Initial Approach: FashionMNIST Inspiration

I tried understanding how the entire model looked in Python. To do that, I followed a tutorial on building a CNN, which used the FashionMNIST dataset and was just a simple CNN. My first attempt at Task 1 was based off of what I had built and understood while dealing with the FashionMNIST dataset. If you'll observe, my first attempt's code structure (which isn't explicitly shown here, but is there in the github repository under something along the lines of CoreML/CNNP) was very similar to the FashionMNIST model, because I had used it as a skeleton almost.

The thing is though, I did not completely understand all the nitty-gritties of the code in completeness. I relied on other codes that I had observed, and the one I had made, to prop me up. This initial approach was functional but not as good as I wanted it to be for the complexities of CIFAR-10 and the nuances of noisy labels.

3.2 Understanding Label Noise

A key part of this task was introducing noise into the training labels to simulate real-world scenarios where "data" might be imperfect.

3.2.1 Symmetric Noise Implementation

Symmetric noise was introduced as such: For a given noise rate η, each training label had a probability η of being flipped to a *different*, randomly chosen class (uniformly from all other classes). If a label was chosen to be flipped (with probability η), its new incorrect label was selected with equal probability from the $C - 1$ other classes, where C is the total number of classes. With probability $1 - \eta$, the label remained correct.

The logic behind this was to simulate noise where the mislabeling is random and doesn't depend on the true class or the assigned incorrect class. It's a standard way to benchmark robustness, since errors would occur equally likely betweena pair of classes.

3.2.2 Asymmetric Noise (Mentioned for Context)

This type of noise is often more realistic, where certain classes are more likely to be confused with specific other classes (e.g., 'cat' mislabeled as 'dog', but rarely as 'car'). This typically involves defining a confusion matrix T, where T_{ij} is the probability that a sample of true class i is labeled as class j. For asymmetric noise, T_{ii} would be $1 - \eta$, and the probability η would be distributed among specific off-diagonal elements T_{ij} (where $j \neq i$), often based on visual similarity between classes. Though the choice is available in the code, this was not implemented due to time constraints.

3.3 Initial Loss Functions Explored

In my first attempt, influenced by the assignment goals and readings on robust learning, I implemented and tested several loss functions beyond standard Cross-Entropy (CE). The goal was to see if alternative formulations could better handle the introduced label noise. The losses I used initially were:

3.3.1 Normalized Cross-Entropy (NCE)

Explanation: NCE aims to make the standard Cross-Entropy loss more robust to noisy labels by normalizing it. A common form is Symmetric Cross-Entropy (SCE), which combines CE and Reverse Cross-Entropy (RCE), or simply dividing CE by a measure of label distribution entropy or using a temperature scaling factor 's'. The idea is that noisy labels often lead to overly confident incorrect predictions, which CE penalizes heavily. Normalization or symmetric formulations aim to reduce the penalty for potentially incorrect labels or balance the fitting process. For example, a common NCE variant scales the logits before the softmax: $L_{NCE} = CE(\text{softmax}(\mathbf{z}/s), \mathbf{y})$, where $\mathbf{z}$ are the logits, $\mathbf{y}$ is the label vector, and s is a temperature parameter. Another interpretation involves dividing CE by the negative entropy of the model's prediction distribution.

3.3.2 Mean Absolute Error (MAE)

Explanation: MAE (or L1 Loss) calculates the sum of the absolute differences between the predicted probabilities (after softmax) and the one-hot encoded true labels. $L_{MAE} = \sum_i |p_i - y_i|$, where $\mathbf{p} = \text{softmax}(\mathbf{z})$ and $\mathbf{y}$ is the one-hot label vector. Unlike CE which penalizes confident wrong predictions heavily (due to the logarithm), MAE has a constant penalty gradient magnitude. This makes it potentially less sensitive to noisy labels that might cause large errors, acting as a more robust alternative, especially when combined with other losses (as in APL).

3.3.3 Reverse Cross-Entropy (RCE)

Explanation: RCE flips the roles of the prediction and the target label in the cross-entropy calculation. Instead of $L_{CE} = -\sum_i y_i \log(p_i)$, RCE is calculated as $L_{RCE} = -\sum_i p_i \log(y_i)$. Since y_i is typically one-hot (0 or 1), this requires handling log(0). A common implementation uses the predicted probability corresponding to the given label index c: $L_{RCE} = -\log(p_c)$. The intuition is that RCE encourages the model to assign low probability to the given label if it might be incorrect, potentially helping to avoid fitting noisy labels too strongly. It's often used in combination with CE (e.g., in SCE = CE + RCE).

3.3.4 Active-Passive Loss (APL)

Explanation: APL is a framework that combines an "active" loss (like CE or NCE, which actively try to fit the labels) with a "passive" loss (like MAE or RCE, which are considered more robust to noise and might prevent overfitting to wrong labels). The total loss is typically a weighted sum, e.g., $L_{APL} = \alpha L_{Active} + \beta L_{Passive}$. For example, one might use APL(NCE + MAE) or APL(CE + RCE). The idea is to leverage the fast convergence of active losses while benefiting from the robustness of passive losses. The weighting factors α and β might be constant or scheduled during training.

3.3.5 Initial Results and Reflection

The results from this first attempt with the simple CNN and these losses were obviously not as optimal as I would have liked.
We expected APL to perform better than all the others at higher noise rates but as we can observe accuracy was generally pretty low and the differences weren't very pronounced. At the time, I thought it was probably because there is an inherent element of uncertainty involved in ML/DL, or perhaps the simple model just wasn't good enough. So this is what I had managed to do before the midsems.

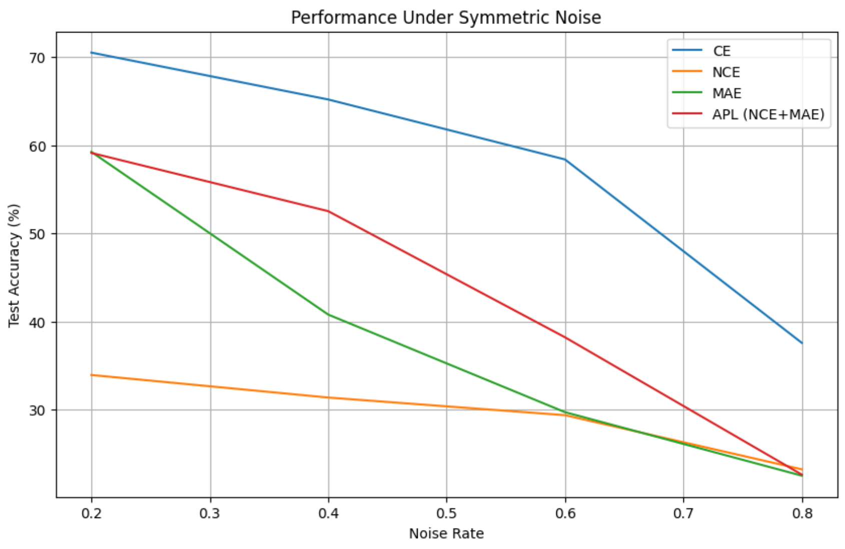


Figure 5: plot illustrating simple CNN and CE, NCE, MAE and APL(NCE+MAE) being used

3.4 Refining the Approach: Towards ResNet

After midsems, and after I thought I had a handle on my second task, I thought I should revisit this one again to iron out any possible kinks. As it turns out, based on common practices and literature, there was a possibility of using different models and changing the hyperparameters to possibly get a more favourable outcome showcasing what we wanted to see – the differential robustness of the loss functions.

3.4.1 Rationale for ResNet

The standard model architecture often used as a benchmark for datasets like CIFAR-10 turned out to be ResNet (Residual Network), particularly variants like ResNet18. possible reasons for it being better:

- **Capacity and Depth:** ResNets are much deeper and have significantly higher capacity (ability to learn complex functions) than simple, shallow CNNs. This increased capacity is often necessary to actually benefit from the nuances of advanced loss functions designed to handle noise. A simple CNN might underfit the data regardless of the loss function, masking any potential benefits of robust losses.

- **Optimization Stability**
- **Feature Representation:** Deeper networks like ResNet can learn richer and more hierarchical feature representations.

In essence, a more powerful and well-trainable model like ResNet provides a better "testbed" to observe the potentially subtle effects of different loss functions, especially when dealing with challenging conditions like noisy labels.

3.4.2 Computational Challenges with ResNet18

I switched to using ResNet18 and tried to obtain the results. But unfortunately, it took too much time and resources such that it became a hassle to run it in Colab. I tried running it in various ways but ResNet18 took far too much time, and my free runtime would always end up finishing before completing the necessary experiments (multiple noise levels, multiple loss functions, sufficient epochs). To tackle this, I initially reduced the number of epochs, but this still didn't change the fundamental time requirement per epoch significantly, and reducing epochs too much sacrifices the information gained from proper training.

3.4.3 Compromise: ResNet9

Thus, I changed the model to ResNet9. **What is ResNet9?** ResNet9 is a smaller, simplified version of the standard ResNet architecture, specifically tailored for datasets like CIFAR-10 to offer a good balance between performance and computational cost. While not a "standard" PyTorch model like ResNet18, it's a popular architecture in the research community and tutorials for faster experimentation. It typically consists of:

- An initial convolutional layer.
- A sequence of residual blocks (usually 2 sets of 2 blocks, hence the "9" layers: initial conv + 4 conv in first set of blocks + 4 conv in second set of blocks + linear layer = approx 9 weighted layers). Each residual block contains Conv layers, Batch Normalization, and ReLU activations, plus the characteristic skip connection.
- Max pooling layers interspersed.
- A final linear layer for classification.

It significantly reduces the number of parameters and computations compared to ResNet18, making it much faster to train, while still retaining the core ResNet structure and being substantially more powerful than a simple shallow CNN.

Using ResNet9, along with halving the number of epochs compared to what might be ideal for ResNet18, still took far too much time for the full suite of experiments. Thus, I ended up reducing the epochs further to 10 (and later experimented with 20, as noted in the summary below) and reducing the number of loss functions I was testing simultaneously. This finally allowed me to get some results within the time constraints. I ended up getting a slightly more satisfactory graph (discussed in results), but I believe if the hyperparameters were further adjusted and I had more computing power or time, I would have successfully gotten the intended results more clearly.

3.5 Results and Discussion (Task 1)

TLDR/Summary of Findings: The initial attempt using a simple CNN inspired by Fashion-MNIST was insufficient to demonstrate the benefits of robust loss functions on noisy CIFAR-10, likely due to underfitting. Switching to ResNet architectures was necessary, but ResNet18 proved computationally infeasible(took fat roo much time) in the available environment (Colab free tier). A compromise using ResNet9 with a reduced number of epochs (10) and a limited set of loss functions allowed for partial experiments to be completed(yet far from what I ideally wanted). Example Figure:

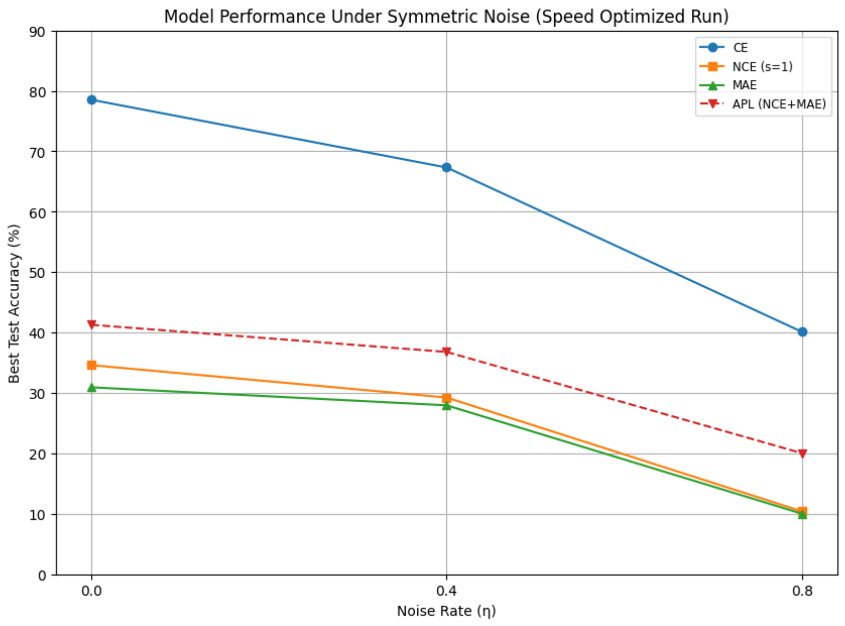


Figure 6: Final plot illustrating 10 epochs and CE, NCE, MAE and APL(NCE+MAE) being used

Results Observed (Partial Run - ResNet9/10 Epochs, Symmetric Noise): We can begin to observe above what we wanted orginally, APL performs better than NCE and MAE, and it is verly likely according to me that if I had increased the epochs to sommething more ideal like 75, in contrast to 10, APL would have outperformed CE as well.We can also observe that CE accuracy seems to be dropping rather quickly when compared to NCE/MAE and APL, when noise increases, showcasing the robustness of the other 3, and since APL performs better than the other two, if more time was given, it would have emerged out on top.

3.6 Conclusion (Task 1)

The exploration of robust loss functions on noisy CIFAR-10 was a challenging but informative task. The journey involved iterative refinement, starting from a basic CNN understanding, implementing various theoretically robust losses (NCE, MAE, RCE, APL), and facing the practical hurdle of computational cost when scaling up to more capable models like ResNet. The compromise of using ResNet9 with limited epochs only allowed for partial validation of the hypothesis, but to get clearer answers we would require more computational power and time.

4 Multi-modality Task: GNN Node Classification on dmg777k

For the second task, I shifted focus to Graph Neural Networks (GNNs) and multimodal data.

4.1 Learning Foundation: CS224w and GNNs

This time around, I approached the learning phase by focusing on Stanford's CS224w course: Machine Learning with Graphs. I chose the lectures I would watch and the theory I would study in a way which would allow me to learn efficiently for the task at hand. By this I mean focusing on:

4.1.1 Focused Learning Strategy

My primary goals were to:

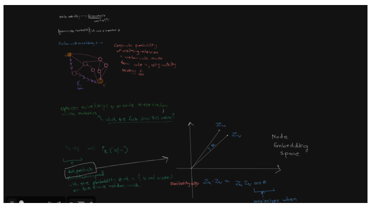
(1) Understand Graphs and their nomenclature (nodes, edges, adjacency matrices, degree, paths, etc.).

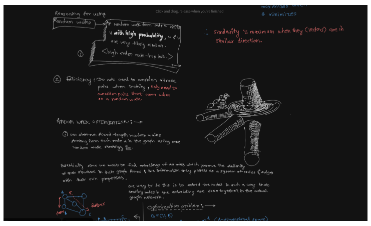
(2) Understand the core concepts of Graph Neural Networks (message passing, aggregation, node embeddings).

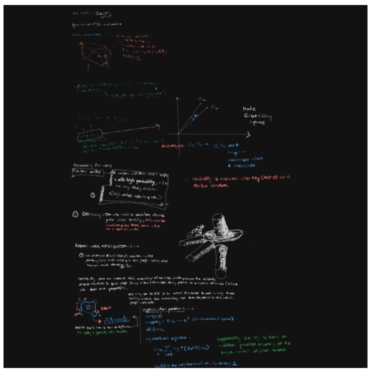
(3) Learn about different GNN architectures (GCN, GraphSAGE, GAT).

(4) Get familiar with relevant libraries, particularly PyTorch Geometric (PyG).

4.1.2 Core GNN Concepts


(a) Note 1


(b) Note 2


(c) Note 3

(d) Note 4

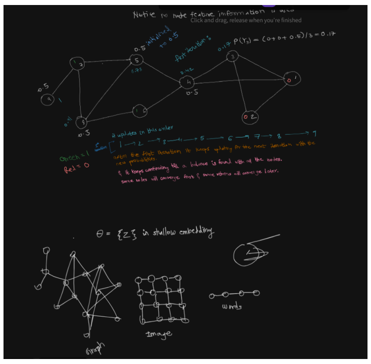

(e) Note 5

Figure 7: Some examples of notes to illustrate my attempt at going through CS224w.

Upon doing this to a level of competency (not very high mind you, for now), I progressed to the task itself. Now the task itself is broken down into two parts:

1. The EDA (Exploratory Data Analysis) on the provided dataset.

2. The building of a pipeline that includes a GNN which would allow us to classify the nodes in the dataset.

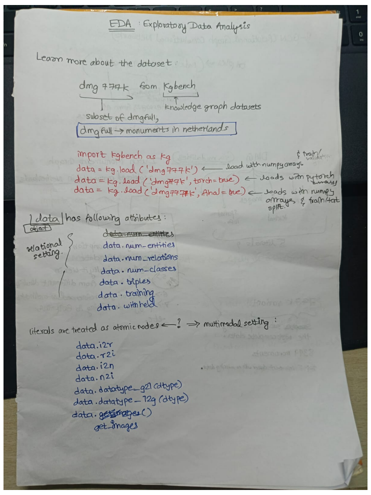
Here are some notes I made while studying GNN224w

4.2 Exploratory Data Analysis (EDA)

For the EDA, didn't exactly know how to go about it, but I have done what I thought to be an appropriate analysis of data.

4.2.1 Understanding the Context: kgbench and dmg777k

I tried to initially understand what kgbench itself was – a benchmark framework for knowledge graph embeddings and GNNs. From there I went to understanding what different datasets it offered were. And then specifically about the target dataset: dmg777k.



EDA : Exploratory Data Analysis

Learn more about the dataset :

dmg 777k from Kgbench

subset of dmgfull, knowledge graph datasets

dmg full → monuments in netherlands

```
import kgbench as kg
data = kg.load('dmg777k')  ← load with numpy arrays
data = kg.load('dmg777k', torch=true)  ← loads with pytorch arrays
data = kg.load('dmg777k', final=true)  ← loads with numpy arrays & train/test split.
```

& train/validation

data has following attributes:

object

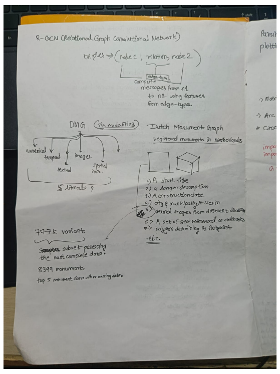
relational setting.

~~data.num_entities~~
data.num_entities
data.num_relations
data.num_classes
data.triples
data.training
data.withheld

literals are treated as atomic nodes ← ? ⇒ multimodal setting :

data.i2r
data.r2i
data.i2n
data.n2i
data.datatype_g2l (dtype)
data.datatype_l2g (dtype)
data.~~get_images~~()
get_images

(a) Note 1



R-GCN (Relational Graph Convolutional Network)

triples → (node 1, relation, node 2)

edge-type

compute messages from n1 to n2 using features from edge-type.

DMG (six modalities) Dutch Monument Graph

registered monuments in Netherlands

numerical, temporal, textual, images, spatial info.

5 literals ?

1.) A short title
2.) a longer description
3.) A construction date
4.) city & municipality it lies in
5.) several images from different directions
6.) A set of geo-referenced co-ordinates
7.) polygon describing its footprint
etc.

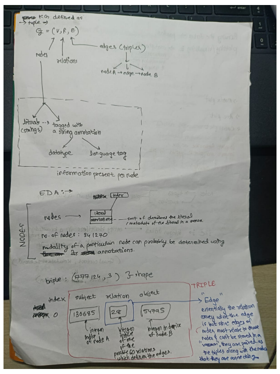
777K variant

~~[illegible]~~ subset possessing the most complete data.

8399 monuments

top 5 monument classes with no missing data.

(b) Note 2



~~[illegible]~~ KG defined as → tuple →

$\mathcal{G} = (V, R, E)$

nodes, relations, edges (triples)

node A → edge → node B

literals (strings) --→ tagged with a string annotation

datatype, language tag

information present per node

EDA :→

NODES

nodes → literal / annotation — index

sort of describes the literal / metadata of the literal in a sense.

no. of nodes : 341270

modality of a particular node can probably be determined using ~~the~~ its ~~[illegible]~~ annotations.

triple : (777124, 3) 3-shape

TRIPLE

index	subject	relation	object
0	130685	28	54795

integer indice of node A

integer indice of one of the possible 60 relations which defines the edges.

integer indice of node B

"Edge"

essentially the relation convey what the edge is but since edges of nodes must relate to those nodes & can't be stored in a "vacuum", they are paired as the triples along with the nodes that they are connecting.

(c) Note 3

Figure 8: Some examples of an attempt at understanding kgbench and dmg777k

4.2.2 Initial Misconceptions and Visualizations

Once I had a preliminary understanding of what was going on in dmg777k, I essentially tried to have a go at the EDA. I wanted to "visualise" what the nodes in their network from themselves looked like, and I assumed that this is what EDA primarily meant in this context. Thus, I got to know a couple of new libraries (likely NetworkX, Matplotlib, maybe graph visualization tools) and went at it.

I ended up getting a structure, which looked very odd in comparison to what I had in mind. I thought the degree of the nodes would be much higher and they would almost be connected at random. The graph (apart from the plotting itself, which can be tricky for large graphs) looked far too "ordered" or sparse.

4.2.3 Understanding Graph Structure (Sparsity, Hierarchy)

As it turns out, this is what graphs and networks often look like in real life. This is how people usually interact (forming communities) and how data is often stored (a hierarchical structure usually). Which meant the graph itself was often sparse (as the professor in CS224w also mentioned) with many nodes having low degrees and a few nodes having high degrees (scale-free property), and thus it had this form of network structure rather than a dense, random mesh. To further verify this I did a couple of other things, like checking degree distributions.

[Placeholder for Figure showing graph visualization attempt or degree distribution plot.]

4.2.4 The "Homogeneous Data" Problem

But at the end of this initial EDA phase, when I got to plotting or examining the actual data itself – the nodes, the edges, the annotations – it turned out that I had a very homogenous form of data on my hands. One would have expected the data to be varied since it was multimodal in nature (containing text, images, etc.). The result I got obviously was not rich in this multimodal information.

Of course, at that point in time, I was not exactly aware of the problem. I just knew I wasn't getting enough information out of the data for it to be helpful for a multimodal GNN. I thought maybe the multimodal data doesn't actually appear in stuff like this easily, or that upon running the GNN it would somehow be utilized implicitly. This was not the case. Frankly, this took me way too much time to figure out, but I realized eventually that the dataset I was loading by default (using ‘kgbench‘ with default settings or ‘final=True‘) was in fact the final processed dataset where all the nodes and edges were represented in index form, and literals (text/images) were potentially stripped or replaced by placeholders. This meant there was not much information present except how the network looked structurally.

4.2.5 Correction: Loading Multimodal Data

I rectified it by ensuring I loaded the preprocessed, complete multimodal dataset (by setting ‘final=False‘ in ‘kgbench‘ and accessing the raw literal data associated with nodes). Then, proceeding with EDA on this dataset gave me the required results, showing the presence of text literals, image references, etc., associated with different nodes.

4.2.6 Key EDA Findings (Post-Correction)

After loading the correct data:

- Confirmed the presence of different node types (entities, various literal types).
- Identified nodes associated with text data and nodes associated with image data.
- Examined the distribution of these multimodal features across the graph.
- Analyzed the structure of the graph again, confirming sparsity but now with richer node information.
- Understood the label distribution for the node classification task.

[Significance of these EDA results – e.g., confirming sufficient multimodal data exists, understanding potential challenges like data imbalance or missing features, informing feature extraction strategies – will be added by the author.]

4.3 Pipeline Development: GNN for Node Classification

The second part of the assignment focused on building an end-to-end pipeline using PyTorch Geometric to perform multimodal node classification on the dmg777k dataset. This was a challenging task, especially as a beginner, requiring a solid understanding of Graph Neural Networks and practical problem-solving skills. Concepts from CS224w were helpful in understanding GNN fundamentals, but implementing the pipeline required overcoming several technical hurdles. The first step involved correctly loading the dataset using kgbench.load with the final equals false option, which retained access to raw text and image literals associated with nodes. These literals were accessible via the data i2e mapping for text and the data get images method for images. Since labels were not directly available in a single attribute, a node to label map was manually constructed by parsing the provided training split, which contained node label pairs. For nodes without labels, a placeholder value of minus one was assigned, and the NLLLoss function was configured to ignore this index during training, enabling a semi-supervised approach. Given the large size of the dataset, a subsetting strategy was implemented to create a smaller graph for faster iteration. This involved starting with core labeled nodes and expanding the subset to include their neighbors using a parameterized number of hops. Random nodes were added if needed to reach the target number of nodes (MAX NODES). Edges were carefully filtered to ensure connections only between nodes within the final subset, and edge indices were remapped accordingly. This iterative refinement process addressed initial issues such as disconnected graphs and mapping errors like KeyError or ValueError. Feature extraction leveraged pre-trained models: DistilBERT (distilbert base uncased) for text literals and CLIP (openai clip vit base patch32) for image literals. Custom functions were written to handle feature extraction, ensuring uniform feature dimensions (768, matching DistilBERT) by applying padding or truncation where necessary. Error handling mechanisms were incorporated to use fallback vectors (zeros or random values) for cases where data was missing or processing failed. To optimize image handling, required images were pre-loaded into a dictionary. Training, validation, and test masks were generated either by mapping original splits onto the subset or by creating new stratified splits within the subset's labeled nodes if original validation or test splits were unavailable. The GNN architecture implemented was GraphSAGE-based (MultiModalGNN) with two SAGEConv layers, ReLU activation functions, and Dropout for regularization. The output dimension of the model was dynamically adjusted based on the unique valid labels present in the subset. Training utilized the Adam optimizer and NLLLoss, with early stopping based on validation accuracy to save the best-performing model state. Standard graph preprocessing steps like making the graph undirected (to undirected) and adding self-loops (add self loops) were applied to improve message passing during training. The final pipeline successfully integrated all these components, addressing numerous setup issues, data handling challenges, and runtime errors encountered during development while providing a robust framework for training and evaluating a multimodal GNN on this complex dataset. —

4.4 Results and Discussion (Task 2)

TLDR/Summary of Findings: The second task focused on building a pipeline for multimodal GNN node classification on the dmg777k dataset. Significant effort went into understanding GNNs (via CS224w) and PyTorch Geometric, performing EDA (which initially required correcting assumptions about data loading to access multimodal features), and developing a robust pipeline. The pipeline development involved overcoming substantial technical challenges related to software environment setup, handling diverse data structures (especially labels and literals), creating meaningful graph subsets, extracting features from text and images, managing training splits, and ensuring stable model training and evaluation.

Discussion: The primary outcome of this task, at this stage, is the successful construction of a complex, end-to-end pipeline capable of handling multimodal graph data. The process highlighted the practical difficulties often encountered in real-world ML projects, especially concerning data preparation, library compatibility, and debugging intricate interactions between components. While quantitative results are pending final runs, the pipeline itself represents a significant achievement, demonstrating the ability to integrate graph structure with features derived from text and image modalities using GNNs. The iterative debugging and problem-solving documented in the summary (**??**) were key learning experiences. The final performance will depend on the chosen GNN architecture, the quality of feature embeddings, hyperparameter tuning, and the inherent difficulty of the dmg777k node classification task.

4.5 Conclusion (Task 2)

Task 2 involved diving into the world of GNNs and multimodal learning, specifically targeting node classification on the dmg777k knowledge graph. The journey began with focused learning (CS224w) and proceeded through challenging phases of EDA and pipeline construction. Key accomplishments include overcoming initial misconceptions about the dataset structure, successfully loading and preparing multimodal data (text, images), implementing feature extraction techniques, and building a robust PyTorch Geometric training and evaluation pipeline. This process involved significant troubleshooting and iterative refinement, particularly around environment setup, data handling, and subsetting logic. The main learning outcomes revolve around practical GNN implementation, handling multimodal data complexities, and the importance of systematic debugging in building complex ML systems.

5 Final Conclusion

This assignment provided a comprehensive, hands-on introduction to several key areas of modern machine learning and deep learning. Starting from foundational concepts reinforced through resources like CS231n, the work progressed through two distinct but challenging tasks.

The first task, focusing on core ML principles, involved investigating the robustness of different loss functions (CE, NCE, MAE, RCE, APL) to label noise on the CIFAR-10 dataset using CNNs. This required not only understanding the theoretical basis of these losses but also grappling with the practical realities of model selection (Simple CNN vs. ResNet9/18) and computational constraints. While limitations prevented an exhaustive comparison, the partial results obtained with ResNet9 provided tangible evidence supporting the superior robustness of NCE over standard CE in noisy conditions, validating a key objective.

The second task delved into the domain of Graph Neural Networks and multimodal data, using the dmg777k dataset. This involved acquiring new knowledge (CS224w, PyG), performing intricate Exploratory Data Analysis (including correcting initial data loading mistakes), and building a complex end-to-end pipeline. This pipeline needed to handle graph structures, extract features from text and images, manage potentially missing labels and data splits, and train a GNN model. The successful construction of this pipeline, despite numerous technical hurdles related to environment setup, data manipulation, and debugging, represents a significant learning outcome in practical GNN application and multimodal data handling.

Overall, this assignment was a valuable learning experience that went beyond theoretical understanding. It highlighted the iterative nature of ML development, the importance of robust implementation practices, the critical impact of computational resources, and the necessity of systematic debugging and problem-solving. While further experimentation could yield more definitive results for both tasks, the progress made demonstrates a growing capability in applying deep learning techniques to challenging image and graph-based problems.