Machine Learning Notes

Last Updated: September 3, 2024

Contents

Unless otherwise specified, images were taken from APS360 slides.

1 Introduction

The following are different machine learning categories:

- **Supervised Learning** - Training a model with labeled data.
- **Unsupervised Learning** - Training a model to make observations without human annotations.
- **Reinforcement Learning** - Training a model by providing rewards from the environment.

Machine learning models can be used to classify objects or solve complicated math problems, but fundamentally, all machine learning models tackle the problem of curve fitting.

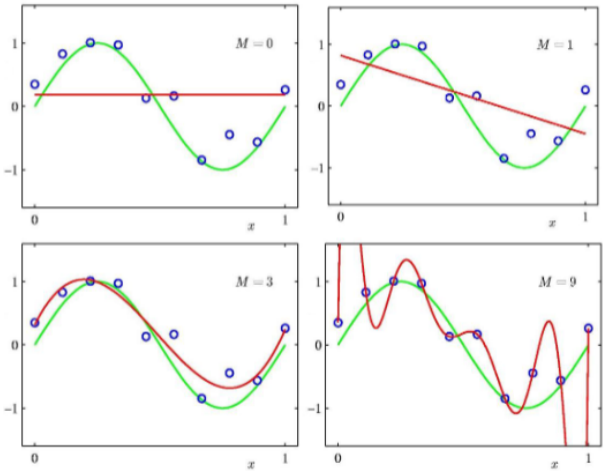

Figure 1: The model tries to fit the points to a curve. The top two curves are examples of underfitting, while the curve in the bottom right is an example of overfitting.

To avoid the problems of overfitting or underfitting, we can split our dataset into a training, validation, and testing dataset. The training dataset is used to train the model, the validation dataset is used to tune hyperparameters, and the testing dataset is used to evaluate the model.

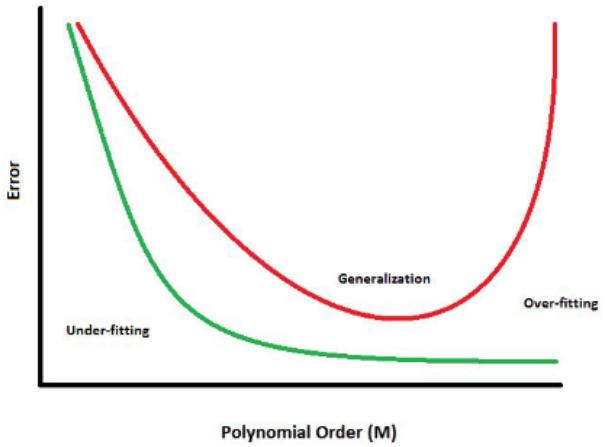

Figure 2: By plotting training (green) and validation (red) loss, we can determine if our model is overfitting or underfitting.

Machine learning models can also be categorized as deterministic or stochastic:

- **Deterministic Models** - These models produce the same output given the same input every time (e.g. feed-forward neural networks).
- **Stochastic Models** - These models incorporate randomness and can produce different outputs given the same input on different runs (e.g. generative networks).

2 Artificial Neural Networks (ANNs)

ANNs are inspired by neurons in our brains. A neuron can be mathematically expressed as

$$y = f\left(w_i x_i + b\right) \tag{1}$$

where y is the output, f is the activation function, w_i is the weight, x_i is the input, and b is the bias. Weights and biases are automatically adjusted during training, and the activation function introduces non-linearity to the model.

2.1 Activation Functions

Below are some common activation functions used in neural networks.

- **Sigmoid** - Used in the output layer for binary classification.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

- **Tanh** - Similar to the sigmoid function, but with a range of $[-1, 1]$.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3}$$

- **ReLU** - Rectified Linear Unit, used in hidden layers.

$$\text{ReLU}(x) = \max(0, x) \tag{4}$$

- **Leaky ReLU** - A variant of ReLU that allows a small gradient when $x < 0$.

$$\text{LeakyReLU}(x) = \max(\text{slope} \cdot x, x) \tag{5}$$

In the output layer, especially for classification problems, usually a softmax function is used to normalize the logits.

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{6}$$

Logits are the outputs of a neural network before the activation function is applied.

2.2 Layers

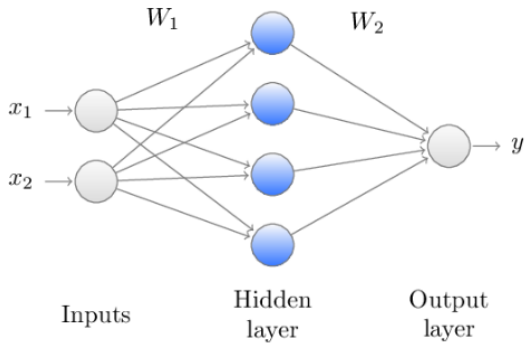

Figure 3: Two Layer Neural Network (Image Source).

In an ANN, neurons are organized into layers.

Usually, there are two layers in an ANN: the input layer, hidden layer, and output layer (**input layer does not count as a layer**).

A two layered neural network **can approximate any function**.

2.3 Parameters

The parameters of an ANN are the weights and biases. The number of weights in a layer can be calculated as,

$$\text{Number of Weights} = \text{Number of Input Connections} \times \text{Number of Output Connections} \tag{7}$$

The number of biases in a layer is equal to the number of output connections.

3 Training ANNs

This section will cover the necessary steps to train an ANN (loss functions, gradient calculations, and optimizers), and it will also provide insight on additional methods such as normalization, regularization, and debugging techniques to improve efficiency of the training process.

In supervised learning, the following steps are taken to train an ANN:

1. Pass the input data through the neural network (forward pass).
2. Calculate the loss between the predicted output and the actual output.
3. Adjust the weights and biases based on gradients and an optimizer (backward pass).
4. Repeat the process until the loss is minimized.

Hyperparameters are parameters that are set before training the model. Some common hyperparameters include the learning rate, number of epochs, batch size, and optimizer.

Usually, the data is grouped into **batches**, so the computer can compute the outputs in parallel to improve efficiency. Small batch sizes may lead to noisy gradients, while large batch sizes will cause the average loss to not change very much leading to the model being stuck in local minima.

The number of **iterations** is the number of passes, each using batch_size number of examples. Parameters are updated once per iteration.

The number of **epochs** is the number of times the model sees the entire training dataset.

3.1 Loss Functions

A loss function computes how bad predictions are compared to the ground truth labels.

- **Mean Squared Error (MSE)** - Used for regression problems.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2 \tag{8}$$

  N is the number of training samples, y_i is the predicted value, and t_i is the true value.

- **Cross Entropy** - Used for classification problems.

$$\text{Cross Entropy} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} t_{i,j}\log(y_{i,j}) \tag{9}$$

  For binary classification, $K = 2$, the equation can be simplified to

$$\text{Binary Cross Entropy} = -\frac{1}{N}\sum_{i=1}^{N} t_i \log(y_i) + (1 - t_i)\log(1 - y_i) \tag{10}$$

3.2 Gradient Calculations

Weights are adjusted based off of gradients.

$$E = (y - t)^2 \qquad f(x) = \frac{1}{1 + e^{-x}}$$

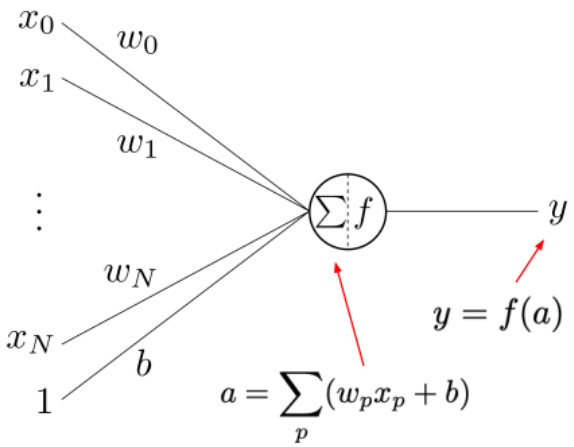

Figure 4: Sample neural network gradient calculation question. E is the loss function, and $f(x)$ is the activation function. We want to determine $\frac{\partial E}{\partial w_p}$.

$$\frac{\partial E}{\partial w_p} = \left(\frac{\partial E}{\partial y}\right)\left(\frac{\partial y}{\partial a}\right)\left(\frac{\partial a}{\partial w_p}\right) \tag{11}$$

$$\frac{\partial E}{\partial y} = \frac{\partial}{\partial y}(y - t)^2 = 2(y - t) \tag{12}$$

$$\frac{\partial y}{\partial a} = \frac{\partial}{\partial a}\left(\frac{1}{1 + e^{-x}}\right) = (1 - y)y \tag{13}$$

$$\frac{\partial a}{\partial w_p} = \frac{\partial}{\partial w_p}(w_p x_p + b) = x_p \tag{14}$$

Thus,

$$\frac{\partial E}{\partial w_p} = 2(y-t)(1-y)(y)(x_p) \tag{15}$$

Instead of calculating the gradients for each sample, we can do mini-batch gradient descent, which averages the loss of a batch per epoch to calculate the gradients.

3.3 Optimizers

An optimizer function determines how much weights are adjusted usually based on the gradients.

For gradient descent, the weights are updated as follows:

$$w_{ji}^{t+1} = w_{ji}^{t} - \alpha \frac{\partial E}{\partial w_{ji}} \tag{16}$$

α is the learning rate, which determines the rate at which the weights are adjusted. A high learning rate will cause the model to jump over minima, while a low learning rate will cause the model to take a long time to converge or cause it to get stuck.

- **Stochastic Gradient Descent (SGD)** - For each iteration, a single sample from the batch is evaluated at random. SGD does more of a global search for the optimum.
- **Mini-Batch Gradient Descent** - For each iteration, a batch of samples is evaluated.
- **Batch Gradient Descent** - For every epoch (there is only 1 iteration per epoch in this case), the entire dataset is evaluated.
- **SGD with Momentum** - Momentum helps SGD perform better in ravines. Each weight has its own velocity.

$$v_{ji}^{t} = \beta v_{ji}^{t-1} - \alpha \frac{\partial E}{\partial w_{ji}} \tag{17}$$

$$w_{ji}^{t} = w_{ji}^{t-1} + v_{ji}^{t-1} \tag{18}$$

- **Adaptive Moment Estimation (Adam)** - Each weight has its own adaptive learning rate and momentum.

$$m_{ji}^{t} = \beta_1 m_{ji}^{t-1} + (1-\beta_1)\frac{\partial E}{\partial w_{ji}} \tag{19}$$

$$v_{ji}^{t} = \beta_2 v_{ji}^{t-1} + (1-\beta_2)\left(\frac{\partial E}{\partial w_{ji}}\right)^2 \tag{20}$$

$$w_{ji}^{t} = w_{ji}^{t-1} - \frac{\alpha}{\sqrt{v_{ji}^{t}} + \epsilon} m_{ji}^{t} \tag{21}$$

3.4 Normalization

Normalization prevents the model from being biased towards certain features with larger ranges.

- **Batch Normalization** - Normalizes the output of a layer batch-wise. The parameters γ and β are learned during training. The number of parameters is equal to $2 \cdot$ Input Size. It increases learning

rate and regularizes the model, but batch normalization cannot work with SGD.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{Batch Mean} \tag{22}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \quad \text{Batch Variance} \tag{23}$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{Normalize} \tag{24}$$

$$y_i = \gamma \hat{x}_i + \beta \quad \text{Scale and Shift} \tag{25}$$

- **Layer Normalization** - Same as batch normalization except it normalizes the output of a layer sample-wise. Not dependent on batch size.
- **Dataset Normalization** - When normalizing the dataset, it is important to determine mean and standard deviation based on the training set and normalize validation and test sets based on the mean and standard deviation of the training set.

3.5 Regularization

Regularization is used to prevent overfitting.

- **Dropout** - During training, we drop activations (set to 0) with probability p. During testing, we scale the activations by $1 - p$ to keep the same distribution.

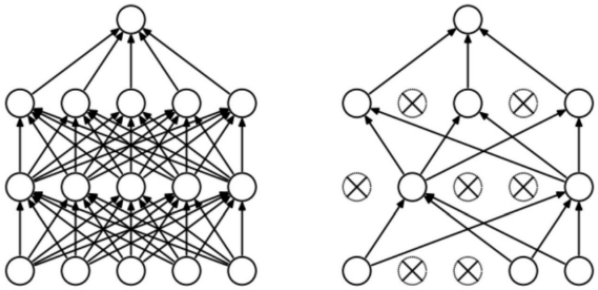

Figure 5: Dropout helps neural networks learn more robust features.

- **Weight Decay (L2 Regularization)** - Prevents weights from growing too much.

$$w_{ji}^{t+1} = w_{ji}^{t} - \lambda \frac{\partial E}{\partial w_{ji}} - \alpha w_{ji}^{t} \tag{26}$$

- **Early Stopping with Patience** - Start a counter when the validation loss starts to increase. If loss decreases again, reset the counter. If the counter reaches a certain threshold, stop training.

3.6 Output Visualization

Confusion matrices can be used to visualize the performance of a classification model.

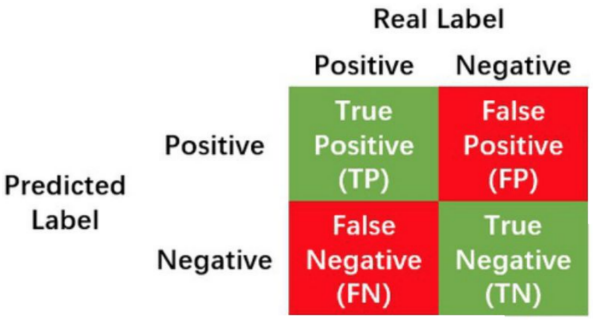

Figure 6: Confusion Matrix for a binary classification problem.

$$\text{Accuracy} = \frac{\sum \text{TP} + \text{TN}}{\sum \text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{27}$$

$$\text{Precision} = \frac{\sum \text{TP}}{\sum \text{TP} + \text{FP}} \tag{28}$$

$$\text{Recall} = \frac{\sum \text{TP}}{\sum \text{TP} + \text{FN}} \tag{29}$$

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{30}$$

The F1 score is the harmonic mean of precision and recall, and it can be used to determine the model's performance.

Other output visualization techniques include the use PCA or t-SNE to visualize high-dimensional data.

4 Convolutional Neural Networks (CNNs)

Artificial neural networks have higher computational complexity, and they do not take into account the spatial relationships between pixels in an image. Convolutional neural networks are used to solve these problems.

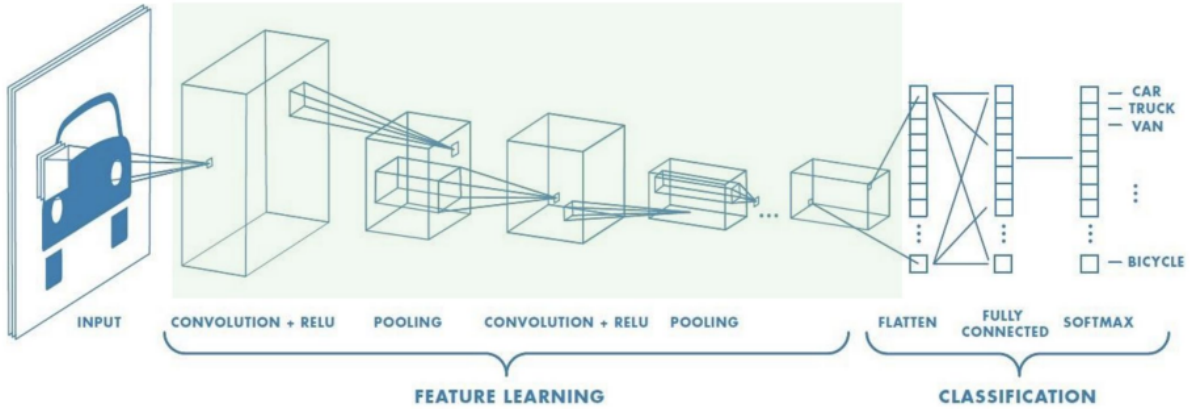

Figure 7: The general architecture of a Convolutional Neural Network.

4.1 Convolutional Layer

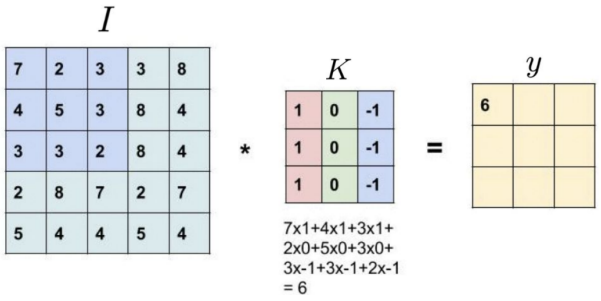

Figure 8: Convolution of image **I** with filter kernel **K**.

A convolution is the process of sliding a filter (**kernel**) containing trainable weights across an image, and taking the dot product to produce a feature map. The earlier layers will learn low level features such as edges or corners while the later layers will learn high level features such as objects.

Terminology:

- **Padding** - Adding zeroes around the border of the image before performing the convolution to keep the output size the same as the input size.
- **Stride** - The number of pixels the filter moves each time.
- **Depth** - The number of channels (e.g. RGB images have three channels: red, green, and blue).
- **Spacial Dimensions** - The height and width of the image.

The output size of the image is

$$\text{Output Size} = \left\lfloor \frac{\text{Input Size} - \text{Kernel Size} + 2 \times \text{Padding}}{\text{Stride}} \right\rfloor + 1. \tag{31}$$

In a convolutional layer, multiple kernels are applied to the input images and summed to produce multiple feature maps. The number of kernels in a convolutional layer is equal to the number of input channels $\times$ the number of output channels.

Thus, the number of weights in a convolutional layer is given by

$$\text{Number of Weights} = \text{Kernel Size} \times \text{Kernel Size} \times \text{Input Channels} \times \text{Output Channels}. \tag{32}$$

The number of bias terms is equal to the number of output channels.

4.2 Pooling Layer

We want to reduce the spatial dimensions of the input to consolidate and remove information not useful for the current task.

One method is to use strided convolutions (stride > 1).

Another method of reducing the number of units before the final output layer is to use pooling layers. Pooling layers have no weights.

$$\text{Output Size} = \left\lfloor \frac{\text{Input Size} - \text{Kernel Size}}{\text{Stride}} \right\rfloor + 1 \tag{33}$$

- **Max Pooling** - Takes the maximum value in the kernel.

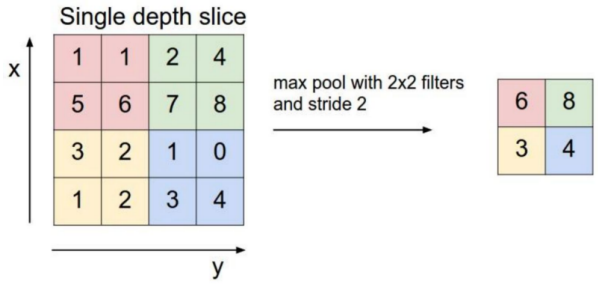

Figure 9: Max pooling provides invariance to small translations of the input.

- **Average Pooling** - Takes the average value in the kernel.

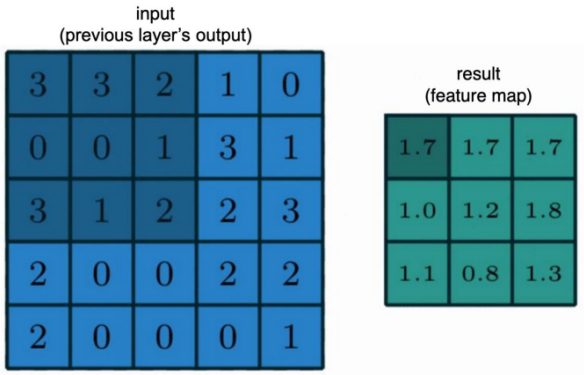

Figure 10: Average pooling is usually worse than max pooling.

4.3 Improving CNNs

- **Data Augmentation** - The creators of AlexNet augmented their training data by applying class preserving transformation (rotations, scale, brightness, mirror) to the ImageNet dataset to increase the size of the training set.
- **Inception Blocks** - GoogLeNet used a mixture of filter sizes per layer called inception blocks. They usually aren't necessary since 3 by 3 filters are more than enough to approximate larger filters as later reported in the VGG paper.
- **1 by 1 Convolutions** - The 1 by 1 convolutions used in GoogLeNet are found to be useful for mapping CNN features into a lower or higher dimensional spaces.
- **Auxiliary Loss** - GoogLeNet was subject to the vanishing gradient problem, so they added intermediate classifiers.
- **Residual Networks** - ResNet used skip connections to prevent vanishing gradients.

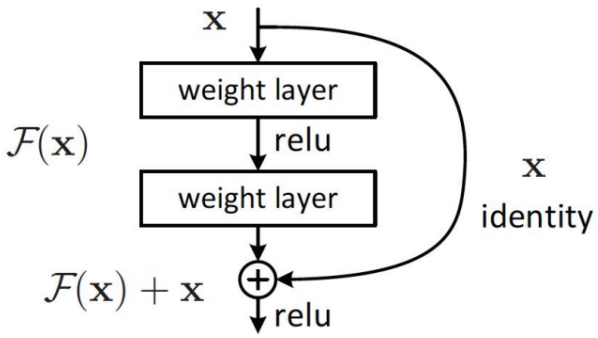

Figure 11: The output of a layer is added to the output of a previous layer.

- **Transfer Learning** - Use a pre-trained model and fine-tune it for a specific task.

5 Autoencoders

Autoencoders are a type of unsupervised learning model that learns to compress data. They consist of an encoder and a decoder.

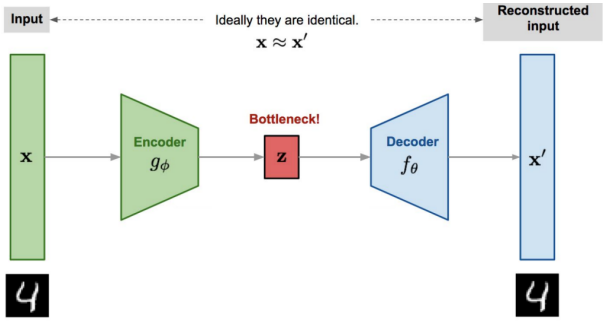

Figure 12: A standard autoencoder consisting of an encoder and decoder which are both fully connected.

An encoder converts the inputs to an internal representation, and a decoder converts the internal representation to the outputs. The autoencoder tries to minimize the difference between the input and output using the mean squared error loss function.

Autoencoders are typically used for dimensionality reduction, pre-training, denoising, data generation, and anomaly detection.

Noise may also be added to the input to train the autoencoder to remove noise. This is known as a denoising autoencoder.

5.1 Variational Autoencoders (VAEs)

To generate new images, we can interpolate between two embeddings and decode those. However, the latent space of a standard autoencoder may not be continuous. Variational autoencoders can be used to solve this problem by sampling from a normal distribution.

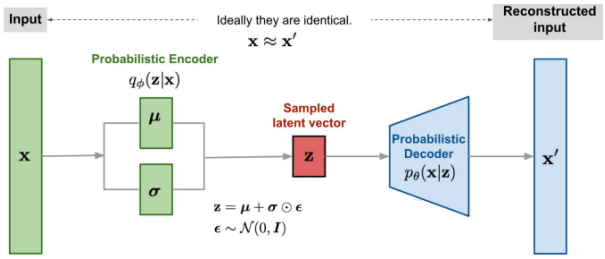

Figure 13: A variational autoencoder consists of a probabilistic encoder and decoder.

We want the encoder distribution $q_\phi(z|x) = \mathcal{N}(\mu, \sigma)$ to be as close to the prior standard normal distribution $p(z) = \mathcal{N}(0, 1)$ as possible.

Kullback-Leibler (KL) divergence can be used to measure the difference between the two distributions.

$$D_{KL}(P||Q) = \sum_{x \in X} p(x) \log \left(\frac{p(x)}{q(x)} \right) \tag{34}$$

Now we can define one of the loss functions for VAEs as simply the KL divergence between the encoder distribution and the prior distribution.

$$D_{KL}(p|q) = \frac{1}{2} \sum_{i=1}^{N} \left[\mu_i^2 + \sigma_i^2 - \left(1 + \log \left(\sigma_i^2 \right) \right) \right] \tag{35}$$

Of course, the other loss function would be the mean-squared error between the input and output.

5.2 Convolutional Autoencoders

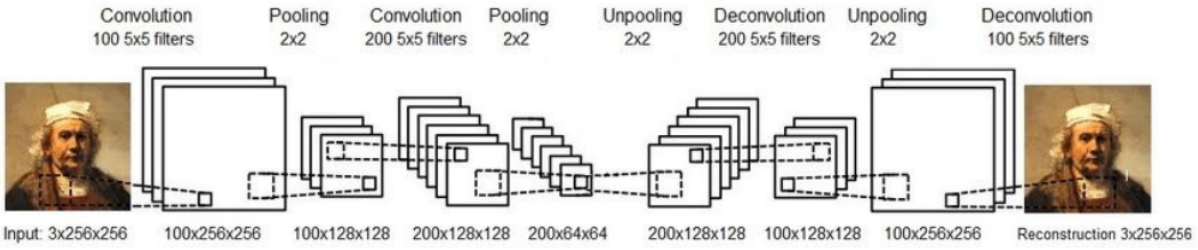

Figure 14: A convolutional autoencoder.

The only difference between a convolutional autoencoder and a standard CNN is that now we must use the transposed convolutions in the decoder.

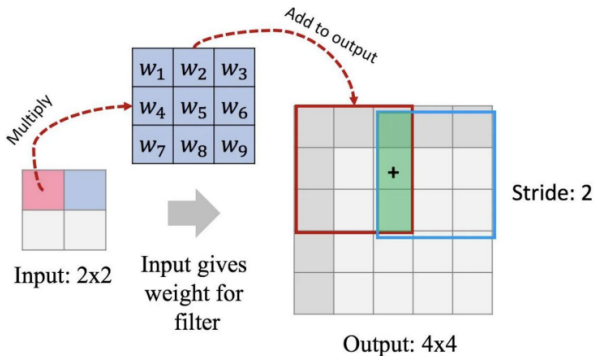

Figure 15: In a transposed convolution, each pixel is multiplied by the kernel and added to the output.

$$\text{Output Size} = (\text{Input Size} - 1) \times \text{Stride} + \text{Kernel Size} - 2 \times \text{Padding} + \text{Output Padding} \tag{36}$$

For transposed convolutions, padding removes rows and columns around the perimeter of the output, and output resolves output shape ambiguity by increasing the output shape on one side.

5.3 Self-Supervised Learning

Self-supervised learning is a type of unsupervised learning where the labels are generated from the data itself.

For example, RotNet rotates images randomly by 0, 90, 180, or 270 degrees, and the model is trained to predict the rotation angle.

Contrastive learning is another self-supervised learning method where samples are contrasted against each other, and the model is trained to minimize the distance between embeddings belonging to the same distribution.

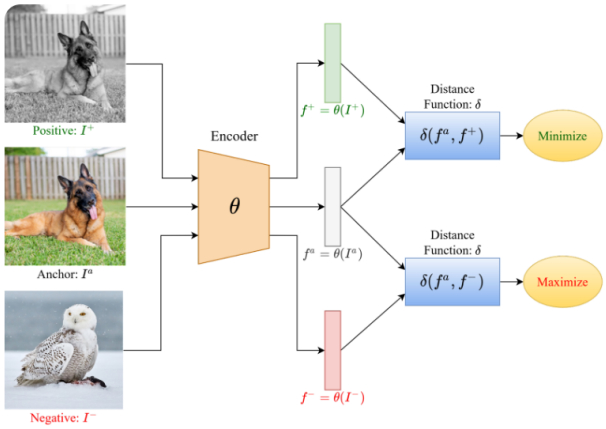

Figure 16: Contrastive learning (Image Source).

6 Recurrent Neural Networks (RNNs)

The concept of RNNs stems from the goal of training a model to understand text.

6.1 Word Embeddings

We must be able to first convert words into a format that is understood a neural network. Converting words into one-hot encodings is not efficient since the vectors are sparse and high-dimensional.

Word2vec is one family of architectures used for solving this issue by converting words into dense, low-dimensional vectors known as word embeddings.

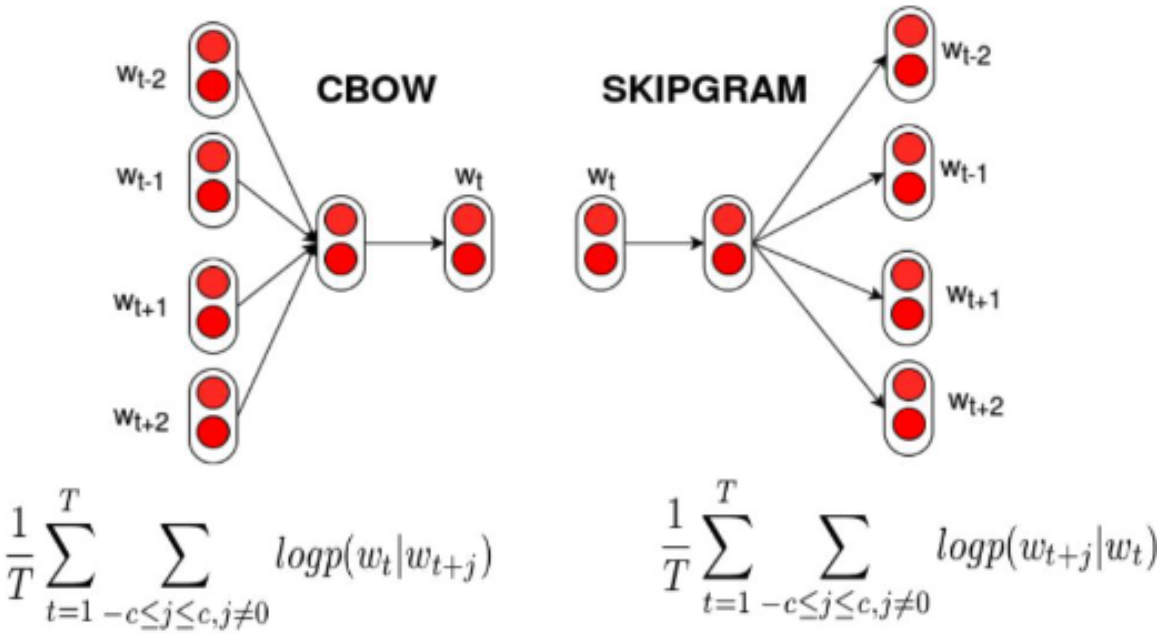

Figure 17: The word2vec architecture includes the Continuous Bag of Words (CBOW) and Skip-gram models.

The skip-gram model predicts the neighboring words of a given word. An n-Gram is a contiguous sequence of n items from a given text. A k-Skip n-Gram is an n-gram that can involve a skip operation of size k or smaller.

At the end of the training process, the hidden layer weights are used as the word embeddings.

The CBOW model predicts a single word based on the surrounding words.

The skip-gram model works well with small datasets, has better semantic relationships (cat and dog), and better representation of infrequent words. The CBOW model is faster to train, has better syntactic relationships (cat and cats), and is better for frequent words.

Another popular word embedding model is GloVe, which is based on matrix factorization. Word co-occurrence frequency counts are created for each word pair, and the embeddings are learned by minimizing the difference between the dot product of the embeddings and the log of the co-occurrence counts (Read More).

Both word2vec and GloVe generate **non-contextual** embeddings meaning that the same word will have the same embedding regardless of the context.

6.2 RNN Layer

RNNs are used to model sequential data.

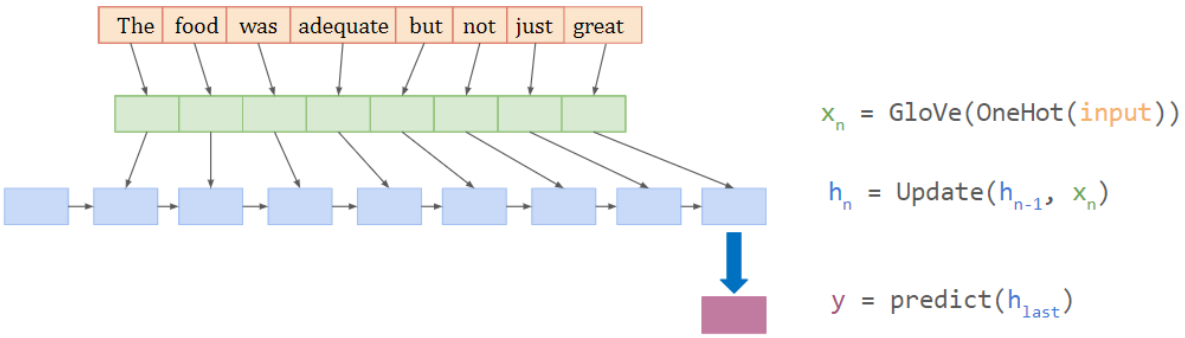

Figure 18: Words would be converted to embeddings and passed into the RNN layer. RNNs are able to learn **contextual** information.

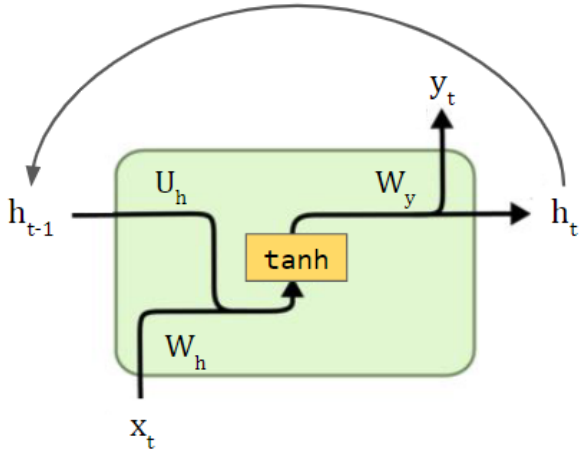

Figure 19: An RNN cell. When each embedding is passed in, the RNN cell decides whether to keep the information within the hidden state or to forget it.

$$h_t = \sigma_h \left(W_h x_t + U_h h_{t-1} + b_h \right) \tag{37}$$

$$y_t = \sigma_y \left(W_y h_t + b_y \right) \tag{38}$$

σ is the sigmoid activation function, x_t is the embedding, h_t is the hidden state, and y_t is the output.

The total number of weights in an RNN cell is given by

$$\text{Number of Weights} = (\text{Input Size} \times \text{Hidden Size} + \text{Hidden Size}^2) \times \text{Number of Layers} \tag{39}$$

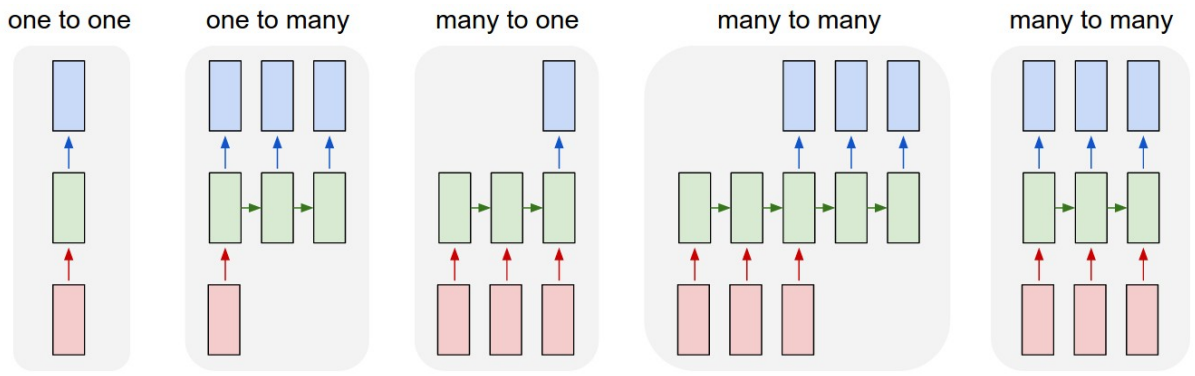

Figure 20: RNNs come in many forms: one-to-one, one-to-many, many-to-one, and many-to-many.

There are two main problems with RNNs: vanishing gradients and exploding gradients. LSTM and GRU cells were developed to solve these problems which will be discussed in the following sections.

6.3 Long Short-Term Memory (LSTM)

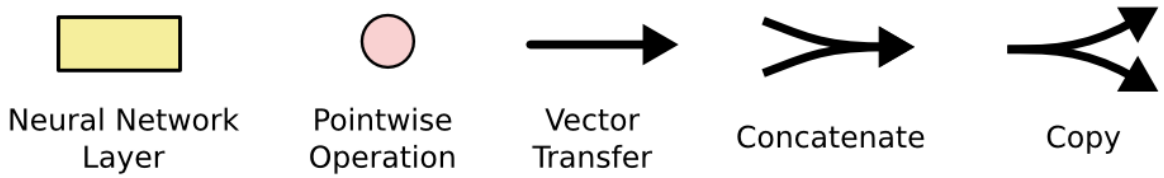

Figure 21: Symbols for RNNs.

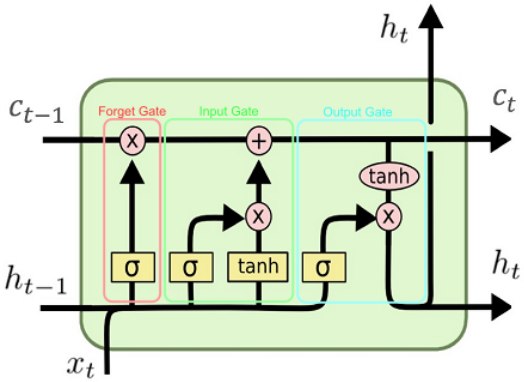

Figure 22: LSTM Cells have a cell state in addition to the hidden state and three gates.

The forget gate determines what information to throw away from the cell state.

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{40}$$

The input gate determines what information to store in the cell state.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{41}$$

$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{42}$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{43}$$

The output gate determines how much of the long-term memory should construct the short term memory.

$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \tag{44}$$
$$h_t = o_t \cdot \tanh\left(C_t\right) \tag{45}$$

6.4 Gated Recurrent Units (GRUs)

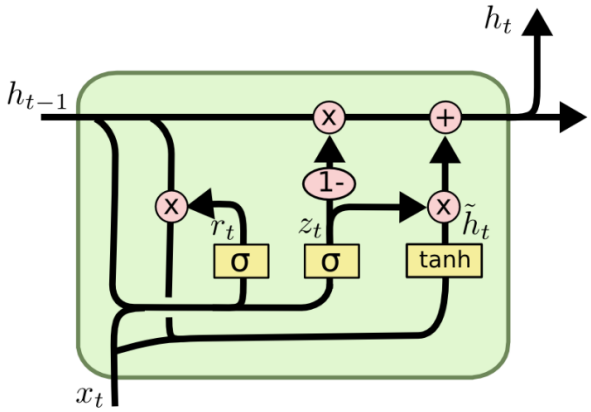

Figure 23: GRUs combines the forget and input gates from LSTMs into a single update gate. It also merges the cell state and hidden state.

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t] + b_z\right) \tag{46}$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t] + b_r\right) \tag{47}$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t \cdot h_{t-1}, x_t] + b\right) \tag{48}$$
$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \tag{49}$$

LSTMs and GRUs are much better at learning long-term relationships compared to standard RNNs.

6.5 Deep and Bidirectional RNNs

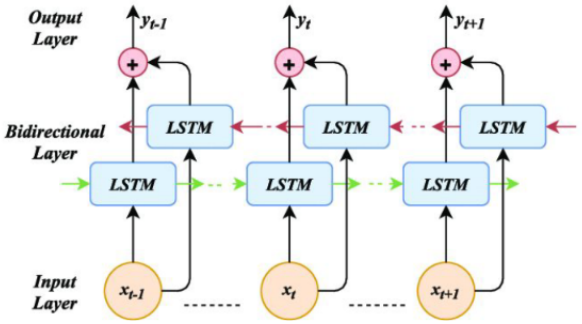

Figure 24: Bidirectional RNNS are useful where a prediction is dependent on both past and future information.

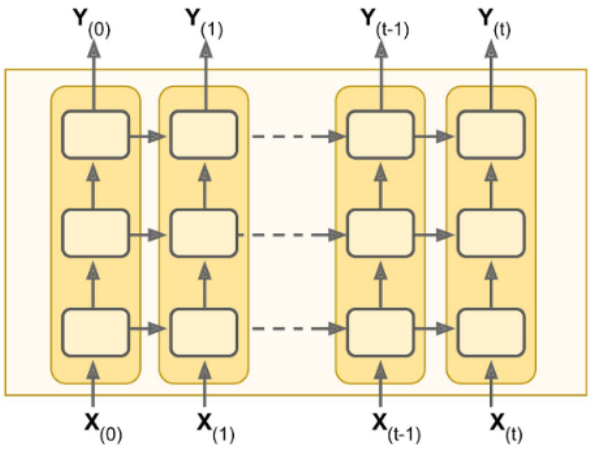

Figure 25: Deep RNNs have multiple layers. Representations in the first layers are better for syntactic tasks while representations in the last layers are better for semantic tasks.

6.6 Sequence to Sequence Models

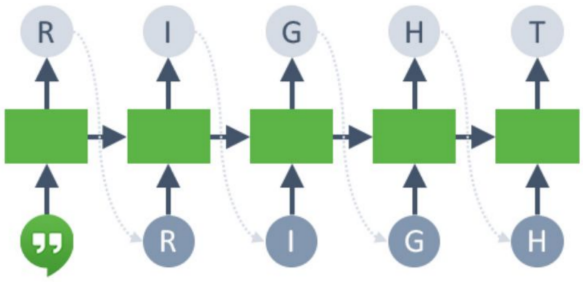

Figure 26: A sequence generating RNN.

There are a few more nuances when generating sequences with RNNs.

The RNN know when to start or stop a sequence with the control symbols ⟨BOS⟩ and ⟨EOS⟩. They are used to indicate the beginning and end of a sequence.

During training, the model is fed the ground truth token at each time step, making training more efficient. This is known as **teacher forcing**.

During Inference, selecting the token with the highest probability will not always work well. It will result in minimal diversity and lots of grammatical errors. The following are three sampling strategies to try out.

- **Greedy Search** - Select the token with the highest probability.
- **Beam Search** - Selects the sequence of tokens with the highest probability.
- **Softmax Temperature Scaling** - Randomly sample the next token based on the softmax distribution (multinomial regression for multiple classes and logistic regression for two classes). A higher temperature will result in higher quality samples but less variety. A lower temperature will result in lower quality samples but more variety.

7 Generative Adversarial Networks (GANs)

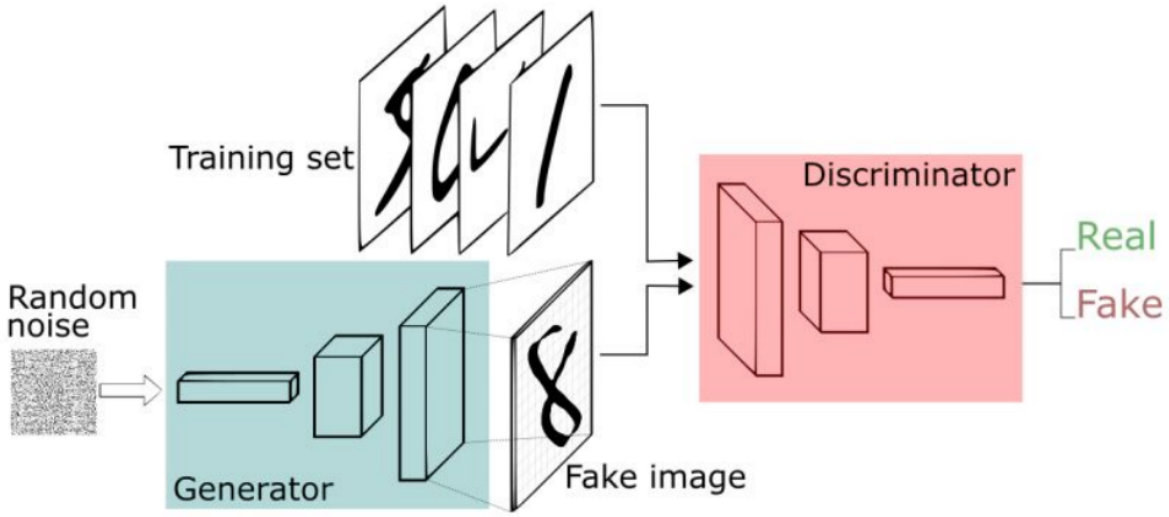

Figure 27: Generative Adversarial Networks consist of a generator and a discriminator. The generator tries to generate realistic images, while the discriminator tries to distinguish between real and fake images.

Binary cross-entropy is used as the loss function for GANs. The weights of the generator are updated based on the discriminator's loss.

```
for each iteration do
    for k steps do
        Sample mini batch of m noise samples and m real samples.
        Update the discriminator by ascending its stochastic gradient.
    end for
    Sample minibatch of m noise samples.
    Update the generator by descending its stochastic gradient.
end for
```

7.1 Problems with GANs

If the discriminator is too good, then the generator will not learn (vanishing gradients).

The generator may produce the same output, and if the discriminator is trapped in a local minimum, the generator will not learn (mode collapse).

It takes a long time to train and converge.

To improve GANs, the following can be done:

- Using Leaky ReLU activation functions
- Batch normalization
- Regularizing discriminator weights and adding noise to the discriminator input

7.2 Applications of GANs

- Converting grayscale to color.

- Style transfer:

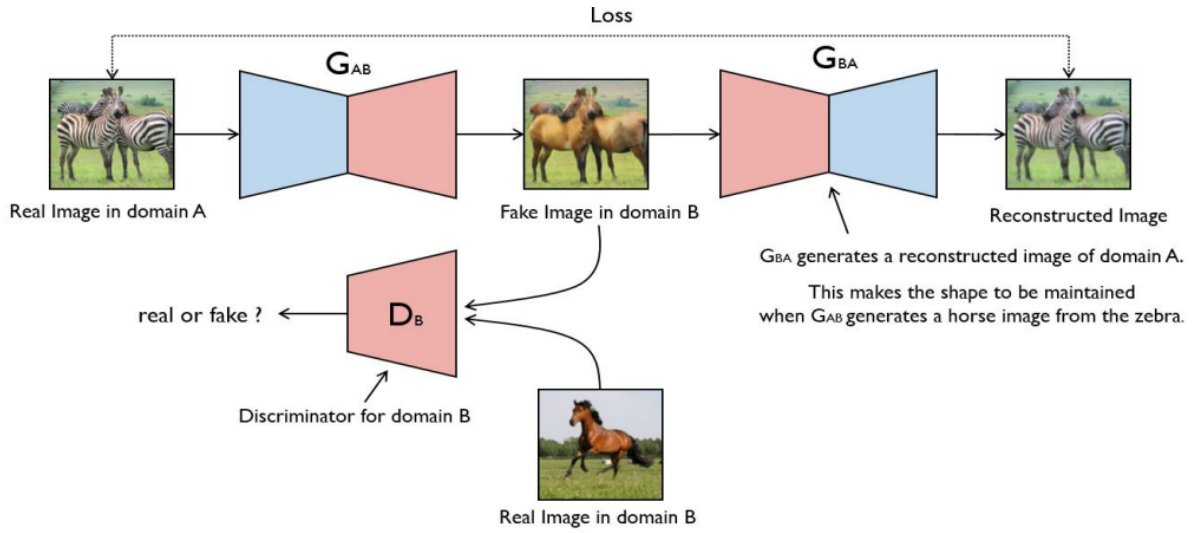

Figure 28: In Cycle GANs, Cycle loss is reconstruction loss between input to cyclegan and output of cyclegan to ensure consistency. There are two generators and two discriminators.

7.3 Adversarial Attacks

The goal of an adversarial attack is to choose a small perturbation ε on an image x so that a neural network f misclassifies $x + \varepsilon$.

Non-Targeted Attacks - Minimize the probability that

$$f(x + \varepsilon) = \text{Correct Class.} \tag{50}$$

Targeted Attacks - Maximize the probability that

$$f(x + \varepsilon) = \text{Target Class.} \tag{51}$$

White Box Attacks - The attacker has access to the model's parameters to optimize ε.

Black Box Attacks - The attacker does not have access to the model's parameters to optimize ε. Substitutes model mimicking target model with a known differentiable function. These attacks often transfer across models.

8 Transformers

RNNs are sequential in nature and can't parallelize the training process. They require large number of training steps, they may have vanishing/exploding gradient problems, and they usually struggle with long range dependencies. Transformers were developed to solve these problems.

Images are from Jay Alammar's Blog.

8.1 Attention

Attention is the fundamental building block of transformers. It allows the model to focus on different parts of the input sequence.

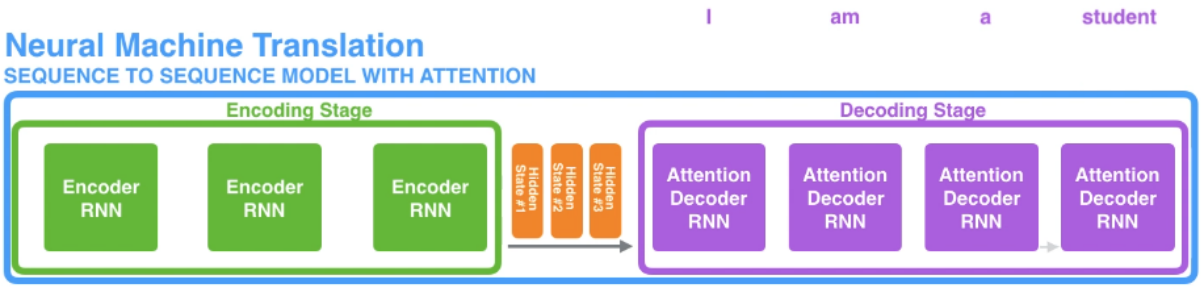

Figure 29: An RNN with attention mechanism.

In an RNN with attention, the encoder passes all hidden states to the decoder.

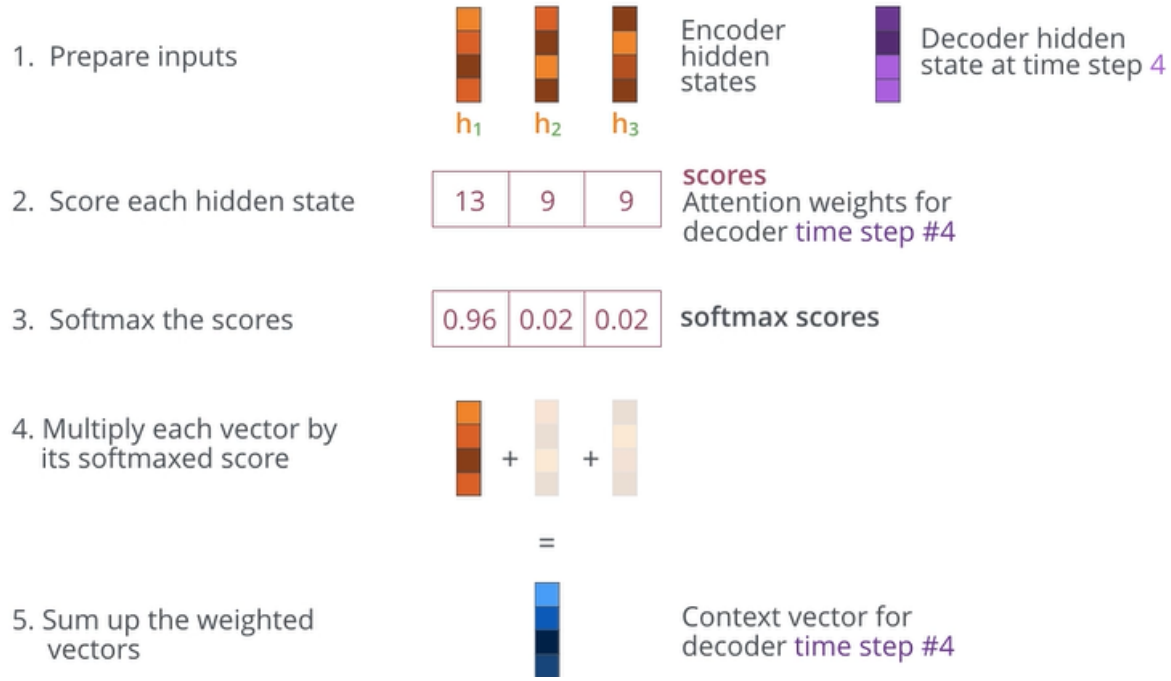

Figure 30: The attention mechanism.

Scores are generated for each word in the input sequence. The scores are multiplied by the hidden states to generate the context vector which is passed into the decoder at every time step. Attention score can be calculated using dot product, cosine similarity, bilinear form, or a feedforward neural network.

Attention Taxonomy:

Cross-Attention combines asymmetrically two separate embedding sequences of same dimension.

Self-Attention is determined by computing the attention weights between a token and all other tokens in the sequence.

8.2 Transformer Architecture

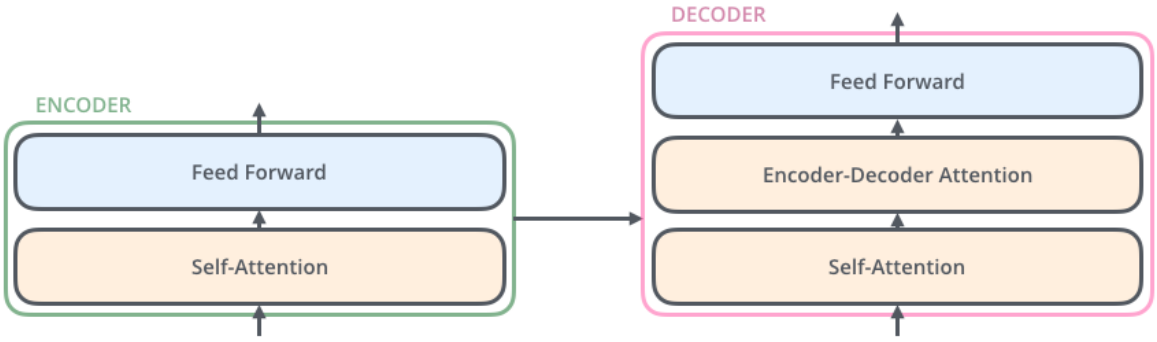

Figure 31: The general architecture of a transformer is composed of multiple encoders feeding into multiple decoders (does not share weights). The encoder and decoder are composed of layers of multi-head self-attention and feedforward neural networks.

Let us first focus on the encoder.

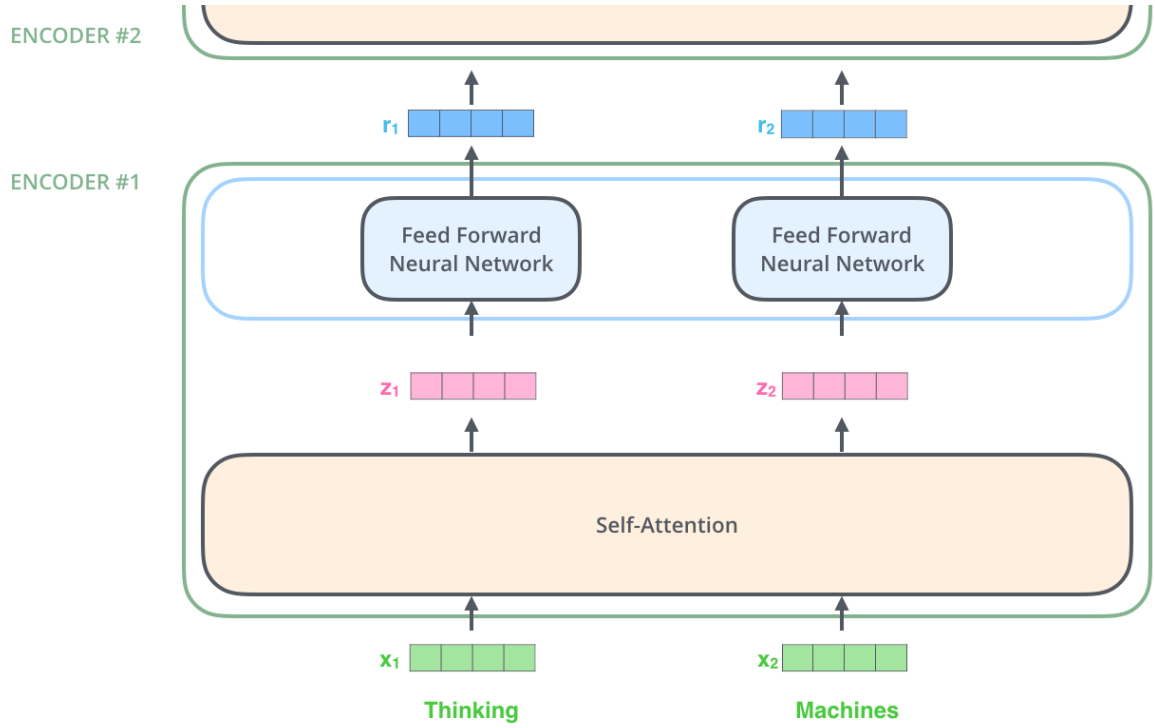

Figure 32: Each word embedding flows through its own path in the encoder.

Attention is implemented using values, keys, and queries matrices which are computed by three linear layers: $Q = XW_Q$, $K = XW_K$, and $V = XW_V$ where $X \in \mathbb{R}^{n \times i}$ is the input embedding.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{k}}\right)V \tag{52}$$

$Q \in \mathbb{R}^{n \times k}$, $K \in \mathbb{R}^{n \times k}$, and $V \in \mathbb{R}^{n \times v}$ where n is the number of tokens, k is the number of keys ($\sqrt{k}$ is known as the attention scaling factor), and v is the number of values. The **attention score matrix** is defined as what is being multiplied by the values' matrix.

Usually multiple **separate** attention layers are run in **parallel** to improve performance. This is known

as **multi-head attention**. The outputs of the attention layers are concatenated and passed through a linear layer, and that output is sent towards the feedforward neural network.

Additionally, a residual connection and layer normalization are applied to both the self-attention layer and feed forward layer.

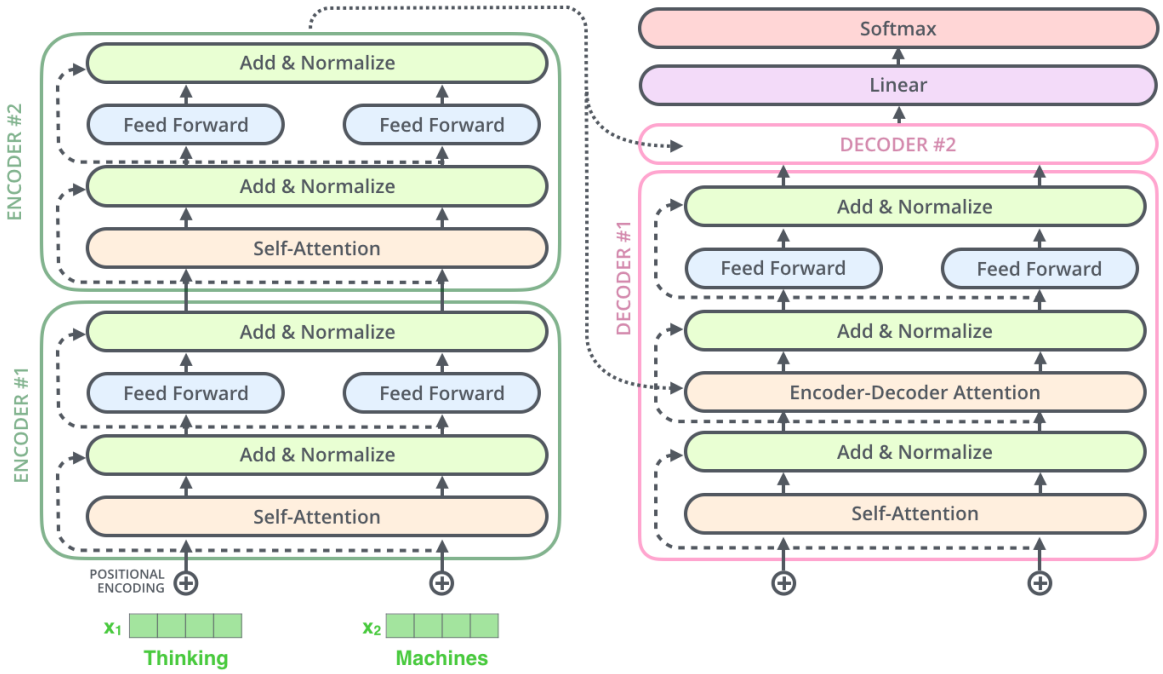

Figure 33: Diagram of the full transformer architecture.

The decoder is similar to the encoder, but it has an additional encoder-decoder layer that uses the encoder's keys and value matrix. A mask operation (setting embeddings to $-\inf$) may also be applied to prevent the model from looking at future tokens.

Lastly, to account for the order of a sequence, positional encodings are added to the input embeddings. Usually, positional encoding matrices are created using sine and cosine functions.

$$PE_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{53}$$

$$PE_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{54}$$

8.3 Applications

- **BERT** - Bidirectional Encoder Representations from Transformers. BERT is a transformer model that requiring deep understanding of the text.

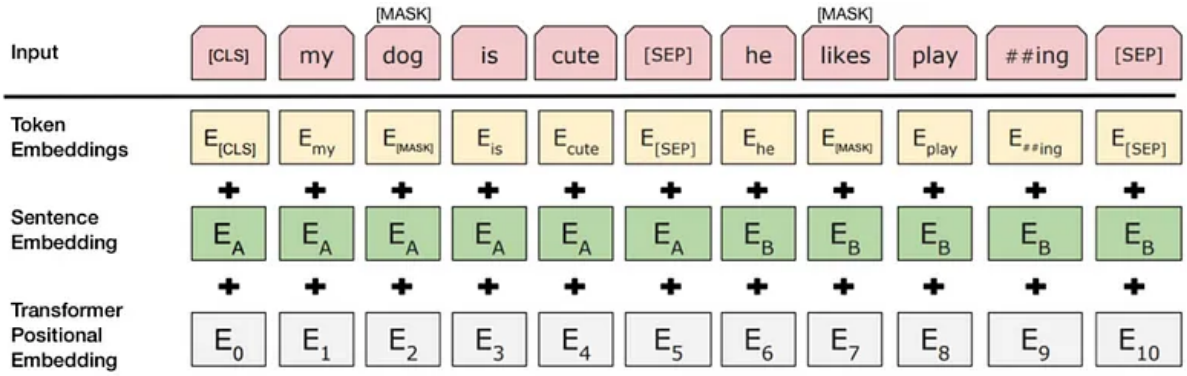

Figure 34: The input embeddings for BERT are the sum of token embeddings, sentence embeddings, and positional embeddings. [CLS] token always appears at the start of the text, and is specific to classification tasks. [SEP] token is used to separate sentences. The sentence embeddings are used to specify which sentence the token belongs to (vector of 0s for the first sentence and 1s for the second sentence).

BERT is trained using masked language modeling and next sentence prediction (multitasking).

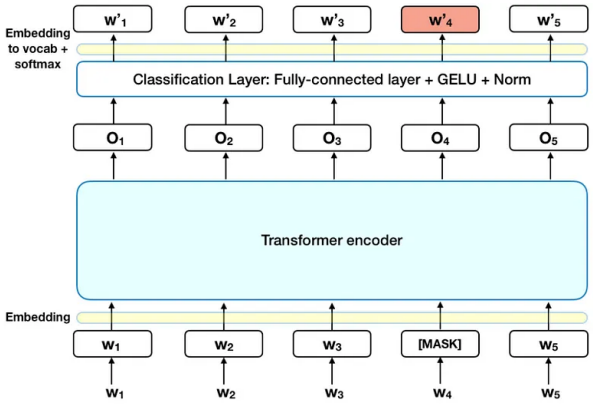

Figure 35: For masked word prediction, 15% of the words are masked using the [MASK] token. GELU stands for Gaussian Error Linear Unit.

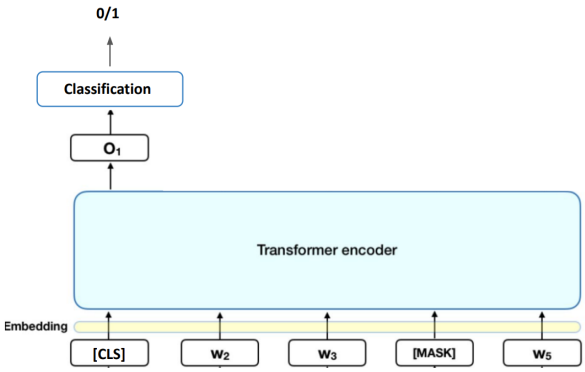

Figure 36: For next sentence prediction, the model is trained to predict whether two sentences appear next to each other in the original text The [CLS] token output is passed to the classifier.

Bert used a combination of semi-supervised learning (BooksCorpus and English Wikipedia) and supervised learning on a labeled dataset (Spam Detection).

- **ViT** - Vision Transformers. ViT applies transformers to images. The image is divided into patches, and the patches are flattened and passed through a linear layer to produce class predictions.

9 Graph Neural Networks (GNNs)

GNNs are used to model graph-structured data such as chemical compounds and social networks. GNNs can classify nodes, classify graphs, and predict edges.

9.1 Sets and Graphs

Sets are collections of unordered elements. There have been several models developed to work with sets, such as the Deep Sets. Essentially Deep Sets is a model which uses a single **shared** neural network to project each item to a shared space. Then, the items are aggregated using an order invariant function (e.g. sum, mean, max). Finally, another neural network is used to predict the output (classification).

A graph is a collection of nodes and edges. The edges can be represented in an adjacency matrix (Dense Implementation). Sparse implementation uses an index vector which maps each node to its respective graph in the batch.

The **degree** of a node is the number of edges connected to it. Like sets, graphs are also order-invariant.

9.2 GNN Architecture

The fundamental operation of all GNNs are message passing and readout.

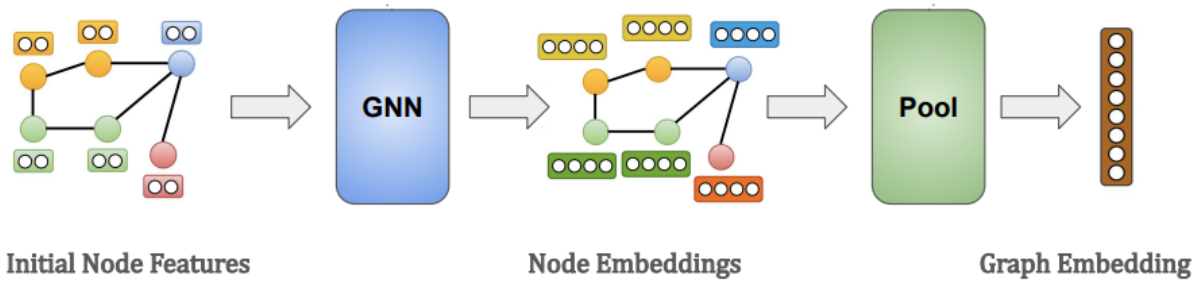

Figure 37: A simple GNN architecture with message passing (GNN) and readout (pool).

Message-passing is applied to each node in the graph. For each, node we combine the aggregated embeddings of its neighboring nodes with its own embedding to produce a new embedding.

$$h_v^{(k)} = \text{COMBINE}\left(h_v^{(k-1)}, \text{AGGREGATE}\left(\left\{h_u^{(k-1)}, h_v^{(k-1)}, e_{uv}\right\}_{u \in N(v)}\right)\right) \tag{55}$$

$N(v)$ is the set of neighboring nodes of v, $h_v^{(k)}$ is the embedding of node v, and e_{uv} is the edge between nodes u and v. The aggregate function must be an order-invariant function such as sum, mean, attention, or max.

Readout aggregates all node embeddings to produce a graph embedding.

$$h_G = \text{READOUT}\left(\left\{h_v^{(k)}\right\}_{v \in G}\right) \tag{56}$$

The different instantiations of aggregation functions define the different types of GNNs.

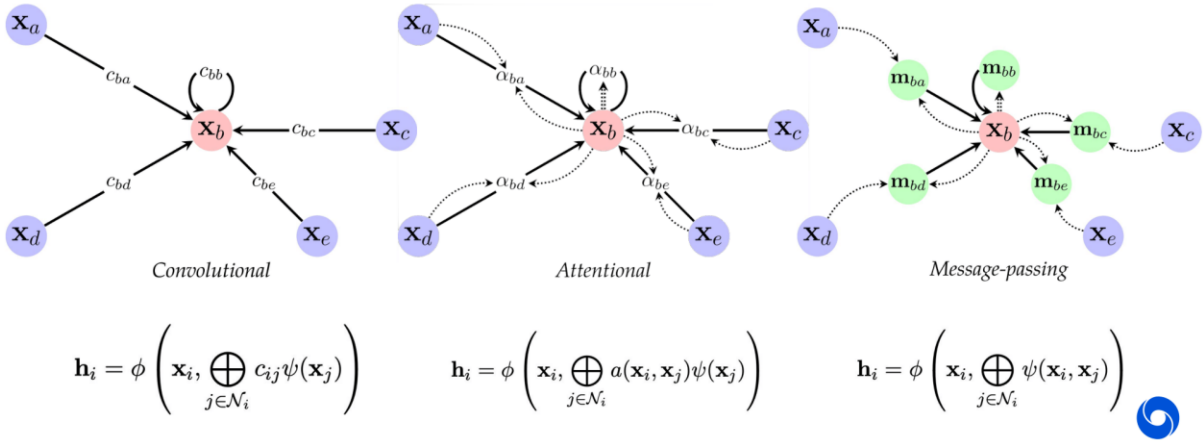

Figure 38: Three types of GNNs.

9.3 Graph Convolutional Networks (GCNs)

The simplest layer of GNN we can define is

$$H = \sigma(AXW) \tag{57}$$

where σ is the activation function, A is the adjacency matrix, X is the feature matrix, and W is the weight matrix.

However, we want to include the node itself into the feature vector sum. To do that we can add the identity matrix to A.

$$A = A + I \tag{58}$$

Furthermore, we also need to normalize A using a diagonal degree matrix D such that all the rows sum to one.

$$A = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \tag{59}$$

To influence the embeddings from further neighborhoods, we can stack multiple layers of GCNs.

9.4 Graph Attention Networks (GATs)

Attention mechanisms can be used to weight the importance of neighboring nodes.

1. A shared neural network is first used to calculate the attention score between two nodes.

$$e_{ij} = NN(h_i, h_j) \tag{60}$$

2. The attention score is normalized using the softmax function.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k\in N(i)} \exp(e_{ik})} \tag{61}$$

3. The normalized attention score is used to calculate the new embedding.

$$h_i = \sigma\left(\sum_{j\in N(i)} \alpha_{ij} W h_j\right) \tag{62}$$