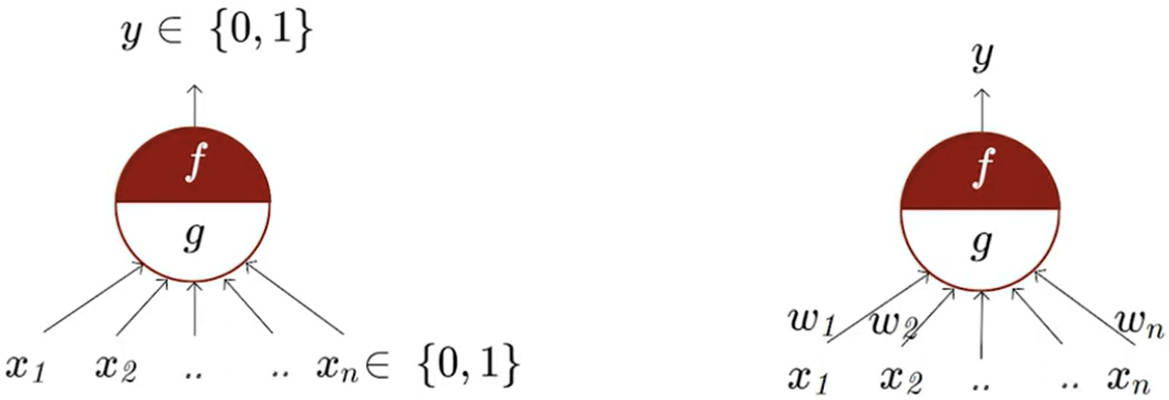
1 History of Deep Learning

1.1 Biological Neurons

- **Reticular Theory**: Nervous system is a single continuous network as opposed to a network of many discrete cells.
- **Staining Technique**: Chemical reaction that allows to examine nervous tissue in much greater detail.
- **Neuron Doctrine**: Nervous system is actually made up of discrete individual cells forming a network.

1.2 From Spring to Winter of AI

- **McCulloch Pitts Neuron**: A highly simplified model of the neuron.
- **Perceptron**: "The perceptron may eventually be able to learn, make decisions, and translate languages."
- "The embryo of an electric computer that the Navy expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."
- Book "**Perceptrons**" by Minsky and Papert outlined the limits of what perceptrons could do.
- **Backpropagation**: First used in the context of artificial neural networks.
- **Universal Approximation Theorem**: A multilayered network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision.



(a) Simple Model of the Neuron

(b) Model of the Perceptron

Figure 1: First models of neurons

1.3 The Deep Revival

- **Unsupervised Pre-Training**: Described an effective way of initializing the weights that allows deep autoencoders networks to learn a low-dimensional representation of data.

1.4 From Cats to Convolutional Neural Networks

- **Hubel and Wiesel Experiment**: Experimentally showed that each neuron has a fixed receptive field -i.e. a neuron will fire only in response to visual stimuli in a specific region in the visual space(Motivation for CNNs).

1.5 Faster, higher, stronger

- **Better Optimization Methods**: Faster convergence, better accuracies. Examples: AdaGrad, RMSProp, Adam, Nadam, etc.
- **Better Activation Functions**: Many new functions have been proposed, leading to better convergence and performance. Example: tanh, ReLu, Leaky Relu, etc.

1.6 The Curios Case of Sequences

- **Sequences**: Each unit in the sequence interacts with other units. Need models to capture this interaction. Example: Speech, Videos, etc.
- **Hopfield Network**: Content-addressable memory systems for storing and retrieving patterns.
- **Jordan Network**: The output state of each time step is fed to the next time step, thereby allowing interactions between time steps in the sequence.
- **Elman Network**: The hidden state of each time step is fed to the next time step, thereby allowing interactions between time steps in the sequence.
- Very hard to train RNNs.
- **Long Short Term Memory**: Solve complex long time lag tasks that could never be solved before(solved the problem of vanishing gradient).
- **Sequence to Sequence Models**: Introduction to Attention!!, However, they were unable to capture the contextual information of a sentence.
- **Transformers**: Introduced a paradigm shift in the field of NLP. GPT and BERT are the most commonly used transformer based architectures.

1.7 Beating humans at their own game

- Human-level control through deep reinforcement learning for playing Atari Game
- **OpenAI Gym**: Toolkit for developing and comprising reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.
- **OpenAI Gym Retro**: A platform for reinforcement learning research on games, which contains 1,000 games across a variety of backing emulators.
- **MuZero**: Masters Go, chess, shogi and Atari without needing to be told the rules, thanks to its ability to plan winning strategies in unknown environments.
- **Player of Games(PoG)**: A general purpose algorithm that unifies all previous approached. Learn to play under both perfect and imperfect information games.

1.8 The rise of Transformers

- **Rule Based Systems**: Initial Machine Translation Systems used handcrafted rules and dictionaries to translate sentences between few politically important language pairs.
- **Statistical MT**: The IBM Models for Machine Translation gave a boost to the idea of data driven statistical NLP, probability based models.
- **Neural MT**: The introduction of seq2seq models and attention. Bigger, hungrier, better models.
- **From Language to Vision**: A vision model based as closely as possible on the Transformer architecture, originally designed for text-based tasks.
- **From Discrimination to Generation**: Sample(Generate) data from the learned probability distribution.Variable Auto Encoders(VAE), Generational Adversarial Networks(GAN), Flow-based models to achieve it.
- GANs don't scale, are unstable and capture less diversity. Diffusion models are one of the alternatives to GAN. Inspired by an idea from non-equilibrium thermodynamics.

1.9 Call for Sanity

- **Paradox of Deep Learning**: Deep learning works so well despite high capacity(susceptible to overfitting), numerical instability(vanishing/exploding gradients), sharp minima(leads to overfitting), non-robustness.
- **Interpretable Machine Learning: A Guide for Making Black Box Models Explainable** by Christoph Molnar.
- **AI Audit challenge**: AI systems must be evaluated for legal compliance, in particular laws protecting people from illegal discrimination. This challenge seeks to broaden the tools available to people who want to analyze and regulate them.

- **Analog AI**: Programmable resistors are the key building blocks in analog deep learning, just like transistors are the core element of digital processors(faster computation).

1.10 The AI revolution in basic Science Research

- **Protein-Folding Problem**: Model proposed to predict protein structure, which would lead to better drug development.
- **Astronomy: Galaxy Evolution**: Predict how galaxies would look like as it gets older.

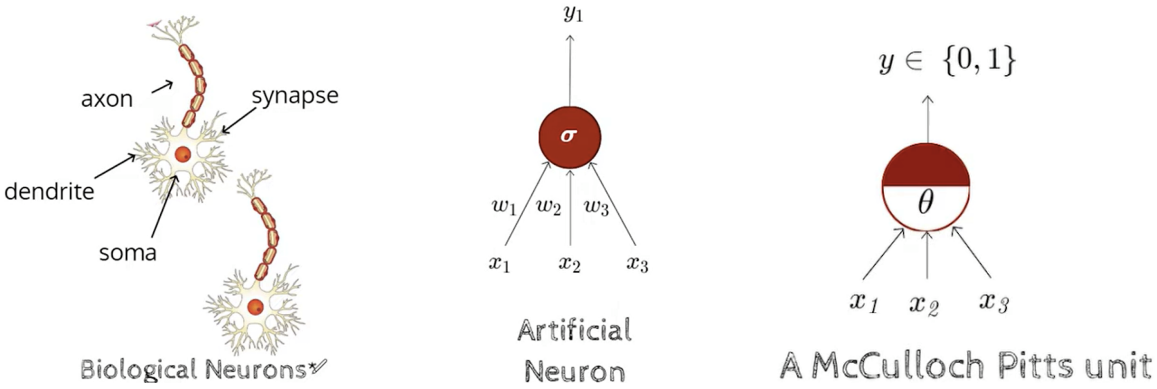
1.11 Efficient Deep Learning

- Build models on small devices, aka phones. Deploying models in resource-constrained devices.

2 Multi Layered Network of Perceptrons

2.1 Biological Neurons

- The most fundamental unit of a deep neural network is called an artificial neuron.
- The inspiration comes from biology(more specifically, from the brain).
- **biological neurons = neural cells = neural processing units**
- **Dendrite**: Receives signals from other neurons
  Synapse: Point of connection to other neurons
  Soma: Processes the information
  Axon: Transmits the output of this neuron.
- This massively parallel network also ensures that there is division of work. Each neuron may perform a certain role or respond to a certain stimulus. The neurons in the brain arranged in a hierarchy.

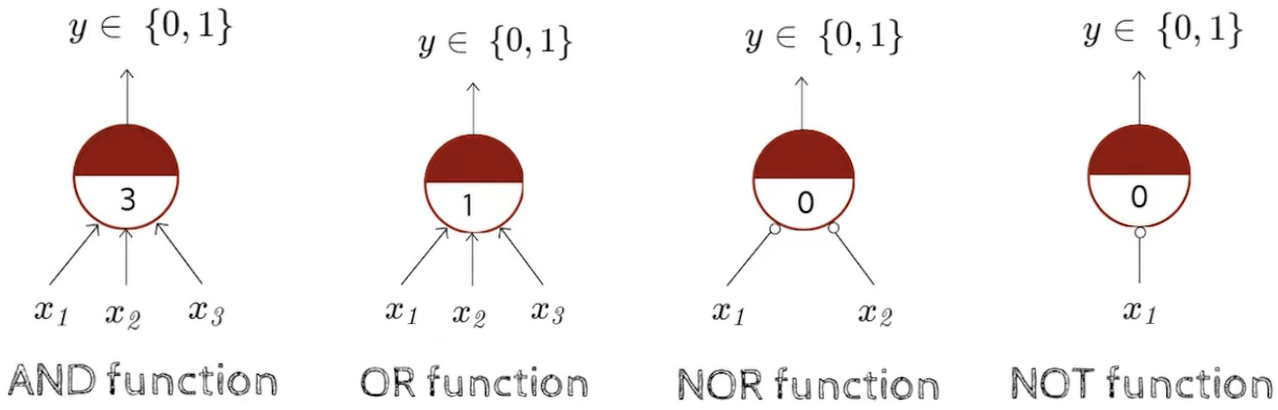


(a) Biological Neuron (b) Artificial Neuron (c) A McCulloch Pitts Unit

Figure 2: Basic Neuron Structure

- **McCulloch Pitts Neuron**: Proposed a highly simplified computational model of the neuron. g aggregates the inputs, and the function f takes a decision based on this aggregation. The inputs can be excitatory or inhibitory.
  Example: $y = 0$ if any x_i is inhibitory, else $g(x_1, ..., x_n) = g(x) = \sum_{i=1}^{n} x_i$
  $y = f(g(x)) = 1$ if $g(x) \geq \theta$ else 0
  θ is called the thresholding parameter.
- circle at the end indicates inhibitory input if any inhibitory input is 1 then the output will be 0.
- Linear separability(for boolean functions): There exists a line(plane) such that all inputs which produce a 1 lie on one side of the line(plane) and all inputs which produce a 0 lie on the other side of the line(plane).
- A single McCulloch Pitts Neuron can be used to represent boolean functions which are linearly separable.

- It can be trivially seen that $\theta = 0$ for the Tautology(always ON) function.

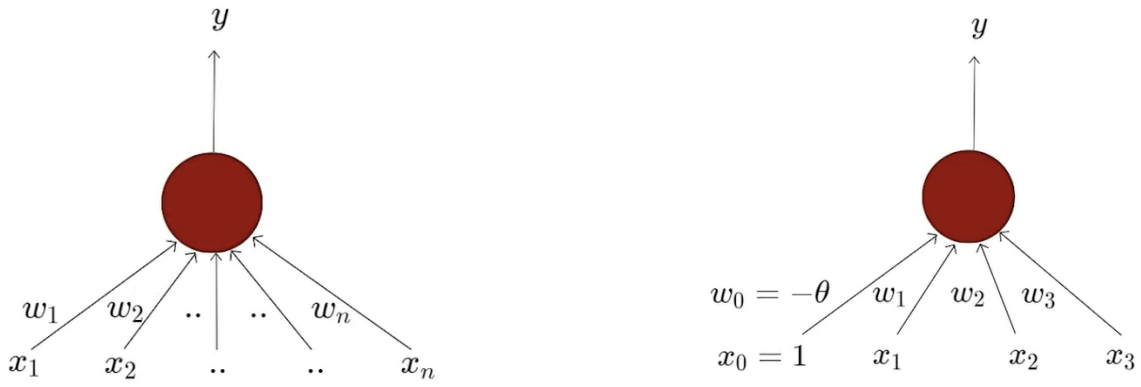


(a) AND function (b) OR function (c) NOR function (d) NOT function

Figure 3: Basic Boolean functions

2.2 Perceptrons

- **Classical Perceptron**: Frank Rosenblatt proposed this model, which was refined and carefully analyzed by Minsky and Papert.
- Main difference is the introduction of numerical weights for inputs and a mechanism for learning these weights.
- $y = 1$ *if* $\sum_{i=1}^{n} w_i \times x_i \geq \theta$ else 0.
  If we take θ on one side and represent $w_0 = -\theta$ and $x_0 = 1$, then we have
  $y = 1$ *if* $\sum_{i=0}^{n} w_i \times x_i \geq 0$ else 0.
- w_0 is called the bias as it represents the **prior**(prejudice).
- From the equation, it can be clearly seen that a perceptron separates the input space into two halves. A single perceptron can only be used to implement linearly separable functions.
- The difference between this and the MP neuron is that now we have weights that can be learned, and the inputs can be real values.



(a) Perceptron (b) Perceptron with w_0

Figure 4: Classical Perceptrons

2.3 Perceptron Learning Algorithm

- **Perceptron learning algorithm**: Consider two vectors $w = [w_0, ..., w_n]$ and $x = [1, x_1, ..., x_n]$, then $w \times x = w^T x$ or the dot product. We can rewrite the perceptron rule as $y = 1$ *if* $w^T x \geq 0$ else 0.
- We are interested in finding the line $w^T x = 0$ which divides the input space into two halves. The angle α between w and any point x which lies on the line must be 90° because $\cos\alpha = \frac{w^T x}{||w||||x||} = 0$. So, the vector w is perpendicular to every point on the line, then it is perpendicular to the line itself.

- For any point such that $w^T x > 0$ we will have $\alpha < 90°$ and for any point such that $w^T x < 0$ we will have $\alpha > 90°$.

Algorithm 1 Perceptron Learning Algorithm

$P \leftarrow$ *inputs with label* 1;
$N \leftarrow$ *inputs with label* 0;
Initialize **w** randomly
while !*convergence* **do**
    Pick random $x \in P \cup N$;
    if $x \in P$ *and* $\sum_{i=0}^{n} w_i \times x_i < 0$ **then**
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
    end if
    if $x \in N$ *and* $\sum_{i=0}^{n} w_i \times x_i \geq 0$ **then**
        $\mathbf{w} = \mathbf{w}$ - $\mathbf{x}$;
    end if
end while
▷ the algorithm converges when all the inputs are classified correctly

- For $x \in P$ if $w^T x < 0$ then it means that the angle α between this x and the current w is greater than $90°$, but we want it to be $< 90°$.
  When we do $w_{new} = w + x$, alpha changes as follows

$$\cos \alpha_{new} \propto (w_{new}^T x) = (w + x)^T x = W^T x + x^T x = \cos \alpha + x^T x$$

$$\cos \alpha_{new} > \cos \alpha \implies \alpha_{new} < \alpha$$

  α_{new} becomes less than, α which is exactly what we wanted, we may not get $< 90°$ in one shot so we keep doing it. Similarly, it can be shown for $x \in N$.

- **Definition**: Two sets P and N of points in an $n-$dimensional space are called absolutely linearly separable if $n+1$ real numbers $w_0, ..., w_n$ exist such that every point $(x_1, ..., x_n) \in P$ satisfies $\sum_{i=1}^{n} w_i \times x_i \geq w_0$ and every point $(x_1, ..., x_n) \in N$ satisfies $\sum_{i=1}^{n} w_i \times x_i < w_0$.

- **Proposition**: If the sets P and N are finite and linearly separable, the perceptron learning algorithm updates the weight vector w a finite number of times. In other words: if the vectors in P and N are tested cyclically one after the other, a weight vector w is found after a finite number of steps t which can separate the two sets.

- **Proof of Convergence**: If $x \in N$, then $-x \in P$ ($\because w^T x < 0 \implies w^T(-x) \geq 0$).
  We can now consider only a single set $P' = P \cup N^-$ and for every element $p \in P'$ ensure that $w^T p \geq 0$.

Algorithm 2 Modified Perceptron Learning Algorithm

$P \leftarrow$ *inputs with label* 1;
$N \leftarrow$ *inputs with label* 0;
$N^- \leftarrow$ *negations of all points in* N;
$P' \leftarrow P \cup N^-$
Initialize **w** randomly
while !*convergence* **do**
    Pick random $p \in P'$;
    if $\sum_{i=0}^{n} w_i \times p_i < 0$ **then**
        $\mathbf{w} = \mathbf{w} + \mathbf{p}$;
    end if
end while
▷ the algorithm converges when all the inputs are classified correctly

Further we will normalize all the p's so that $||p|| = 1$, notice this does not change anything in our step.
Let w^* be the normalized solution vector(we know one exists whose value we don't know).
Now suppose at some time step t we inspected the point p_i and found that $w^T p_i < 0$, then we make correction $w_{t+1} = w_t + p_i$.
Let β be the angle between w^* abd w_{t+1}, then we have $\cos \beta = \frac{w^* w_{t+1}}{||w_{t+1}||}$

$$Numerator = w^* \cdot w_{t+1} = x^* \cdot (w_t + p_i) = w^* \cdot w_t + w^* \cdot p_i \geq w^* \cdot w_t + \delta \ (\delta = \min(w^* \cdot p_i | \forall i))$$
$$\geq w^* \cdot (w_{t-1} + p_j) + \delta \geq w^* \cdot w_{t-1} + w^* \cdot p_j + \delta \geq w^* \cdot w_{t-1} + 2\delta \geq w^* w_0 + (k)\delta \ (By\ Induction)$$

$$Denominator^2 = ||w_{t+1}||^2 = (w_t + p_i) \cdot (w_t + p_i) = ||w_t||^2 + 2w_t \cdot p_i + ||p_i||^2 \leq ||w_t||^2 + ||p_i||^2$$
$$\leq ||w_t||^2 + 1 \leq (||w_{t-1}||^2 + 1) + 1 \leq ||w_0||^2 + (k)$$

So, we have $Numerator \geq w^* \cdot w_0 + (k)\delta$ and $Denominator^2 \leq ||w_0||^2 + (k)$

$$\cos\beta \geq \frac{w^* \cdot w_0 + k\delta}{\sqrt{||w_0||^2 + k}}$$

$\cos\beta$ grows proportional to $\sqrt{k}$
As k(number of corrections) increase $\cos\beta$ can become arbitrarily large, but since $\cos\beta \leq 1$, k must be bounded by a maximum order.
Thus, there can only be a finite number of corrections (k) to w and the algorithm will converge!

2.4 Linearly Separable Boolean Function

- One simple example that is not linearly separable is XOR

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

Table 1: XOR Truth Table

- Most real world data is not linearly separable and will always contain some **outliers**.
- How many boolean functions can we design from 2 inputs.

x_1	x_2	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 2: Functions of 2 inputs

Out of these, only XOR and !XOR are not linearly separable.

- In general, we can have 2^{2^n} boolean functions in n inputs.

2.5 Representation Power of a Network of Perceptrons

- We will assume True = +1 and False = −1, we consider 2 inputs and 4 perceptrons with specific weights. The bias of each perceptron is −2. Each of these perceptrons is connected to an output perceptron by weights (which need to be learned). The output of this perceptron (y) is the output of this network.

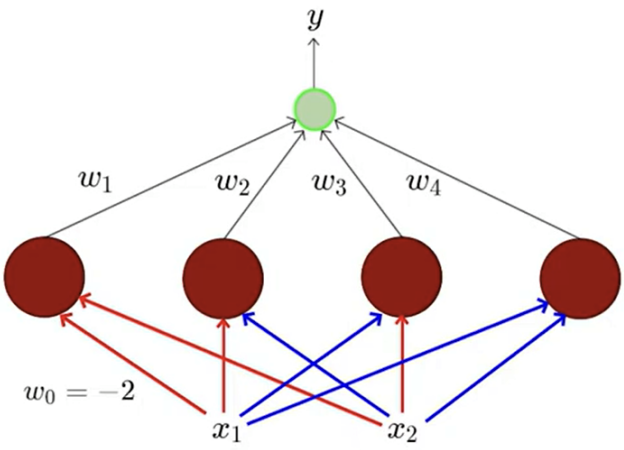


Figure 5: Network of Perceptrons

red edge indicates $w = -1$, and blue edge indicates $w = +1$. x_1 has weights $-1, -1, +1, +1$ and x_2 has weights $-1, +1, -1, +1$ from left to right respectively.

- The above network contains 3 layers.
  The layer containing the inputs (x_1, x_2) is called the **input layer**.
  The middle layer containing the 4 perceptrons is called the **hidden layer**.
  The final layer containing one output neuron is called the **output layer**.
  The output of the 4 perceptrons in the hidden layer are denoted by h_1, h_2, h_3, h_4.
  The red and blue edges are called layer 1 weights
  w_1, w_2, w_3, w_4 are called layer 2 weights.

- This network can be used to implement **any** boolean function linearly separable or not. Each perceptron in the middle layer fires only for a specific input and no perceptrons fire for the same input.

x_1	x_2	XOR	h_1	h_2	h_3	h_4	$\sum_{i=1}^{4} w_i h_i$
0	0	0	1	0	0	0	w_1
1	0	1	0	1	0	0	w_2
0	1	1	0	0	1	0	w_3
1	1	0	0	0	0	1	w_4

Table 3: Truth Table for the Network

This results in four independent conditions.

- In case of 3 inputs we would now have 8 perceptrons in the hidden layer.

- **Theorem**: Any boolean function of n inputs can be represented exactly by a network of perceptrons containing one hidden layer with 2^n perceptrons and one output layer containing 1 perceptron.

- **Note**: A network of $2^n + 1$ perceptrons is not necessary but sufficient.

- **Catch**: As n increases the number of perceptrons in the hidden layers increases exponentially.

- We care about boolean functions, because we can model real world examples into boolean functions or classification problems.

- The network we saw is formally known as Multilayer Perceptron(MLP) or more appropriately "Multilayered Network of Perceptrons".

3 Sigmoid Neurons

3.1 What are they?

- The thresholding logic used by a perceptron is very harsh! It is a characteristic of the perceptron function itself which behaves like a **step function**.

- For most real world applications we would expect a smoother decision function which gradually changes from 0 to 1.

- Instead we would use sigmoid function

$$y = \frac{1}{1 + \exp(-(\omega_0 + \sum_{i=1}^{n} \omega_i x_i))}$$

- We no longer see a sharp transition around the threshold ω_0. Also, y now takes any real value in $[0, 1]$.

- This value can also be interpreted as probability.

3.2 Typical Supervised Machine Learning Setup

- **Data**: $\{x_i, y_i\}_{i=1}^{n}$, we have n data point where x_i is a vector of $\mathbb{R}^m$. Assume $y = \hat{f}(x; \theta)$.

- **Model**: Our approximation of the relation between x and y.

$$\text{For example, } \hat{y} = \frac{1}{1 + e^{-w^T x}} \text{ or } \hat{y} = w^T x \text{ or } \hat{y} = x^T W x$$

- **Learning algorithm**: An algorithm for learning the parameters w of the model.
- **Objective/Loss/Error function**: To guide the learning algorithm. One possibility is $\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$.
- The learning algorithm should aim to minimize the loss function.

3.3 Learning Parameters

- For ease of explanation we will take $f(x) = \frac{1}{1+e^{-(wx+b)}}$, only a single parameter.
- Assume input for training data is $\{x_i, y_i\}_{i=1}^{N} \rightarrow N$ pairs of (x, y).
- Training Objective: Find w and b such that $\mathcal{L}(w, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i))^2$ is minimized.
- **Guess Work(Infeasible)**: Intuitively guess what should be the values of w and b. May never end and is not feasible for writing algorithms.
- **Update rule**: $\theta_{new} = \theta + \eta\Delta\theta$, how to find $\Delta\theta$?
- **Taylor Series**: A way of approximating any continuously differentiable function $\mathcal{L}(w)$ using polynomials of degree n. The higher the degree the better the approximation!

$$\mathcal{L}(w) = \mathcal{L}(w_0) + \frac{\mathcal{L}'(w_0)}{1!}(w - w_0) + \frac{\mathcal{L}''(w_0)}{2!}(w - w_0)^2 + ...$$

- Linear approximation is the first order approximation of the Taylor series. Quadratic approximation is the second order approximation of the Taylor series.
- **Key point**: You can only do this for a very small $\Delta = w - w_0$.
- Let's assume $\Delta\theta = u$, then from Taylor series we have

$$\mathcal{L}(\theta + \eta u) = \mathcal{L}(\theta) + \eta u^T \nabla_\theta \mathcal{L}(\theta) + \frac{\eta^2}{2!}u^T \nabla_\theta^2 \mathcal{L}(\theta)u + ... = \mathcal{L}(\theta) + \eta u^T \nabla_\theta \mathcal{L}(\theta)$$

- η is typically small, so $\eta^2, \eta^3, ... \rightarrow 0$
- Now, we want new loss less than the current loss

$$\mathcal{L}(\theta + \eta u) - \mathcal{L}(\theta) < 0 \implies u^T \nabla_\theta \mathcal{L}(\theta) < 0$$

- Let β be the angle between u^T and $\nabla_\theta \mathcal{L}(\theta)$ and $k = \|u^T\|\|\nabla_\theta \mathcal{L}(\theta)\|$, then we can write

$$-1 \leq cos(\beta) = \frac{u^T \nabla_\theta \mathcal{L}(\theta)}{\|u^T\|\|\nabla_\theta \mathcal{L}(\theta)\|} \leq 1 \implies -k \leq u^T \nabla_\theta \mathcal{L}(\theta) \leq k$$

- $u^T \nabla_\theta \mathcal{L}(\theta)$ is most negative when $cos(\beta) = -1$ or $\beta = 180°$
- **Parameter Update Rule**: From the above we can now say

$$w_{t+1} = w_t - \eta \nabla w_t$$
$$b_{t+1} = b_t - \eta \nabla b_t$$

- We can now write the gradient descent algorithm for our problem as follows

Algorithm 3 Gradient Descent

GradientDescent
$t \leftarrow 0$
$max\_iterations \leftarrow 1000$
$w, b \leftarrow$ initialize randomly
while $t < max\_iterations$ **do**
    $w_{t+1} \leftarrow w_t - \eta \nabla w_t$
    $b_{t+1} \leftarrow b_t - \eta \nabla b_t$
    $t \leftarrow t + 1$
end while

- where ∇w and ∇b are defined for sigmoid neuron as

$$\nabla w = \frac{\partial \mathcal{L}(x)}{\partial w} = (f(x) - y)\frac{\partial}{\partial w}(\frac{1}{1 + e^{-(wx+b)}}) = (f(x) - y) \times f(x) \times (1 - f(x)) \times x$$

$$\nabla b = \frac{\partial \mathcal{L}(x)}{\partial b} = (f(x) - y) \times f(x) \times (1 - f(x))$$

3.4 Representation Power of a multiplayer network of sigmoid neurons

- A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision.

4 Feed Forward Neural Networks

4.1 Structure of Feed Forward Neural Network

- The input to the network is an $n-$dimensional vector.
- The network contains $L-1$ hidden layers having n neurons each. Value of n could be different in each layer.
- Finally, there is one output layer containing k neurons, corresponding to $kclasses$.
- Each neuron in the hidden layer and output layer can be split into two parts: pre-activation and activation(a_i and h_i are vectors).
- The input layer can be called the $0-$th layer and the output layer can be called the $L-$th layer.
- $W_i \in \mathbb{R}^{n \times n}$ and $b_i \in \mathbb{R}^n$ are the weight and bias between layers $i-1$ and i $(0 < i < L)$.
- $W_L \in \mathbb{R}^{k \times n}$ and $b_L \in \mathbb{R}^k$ are the weight and bias between the last hidden layer and the output layer.
- The pre-activation at layer i is given by

$$a_i(x) = b_i + W_i h_{i-1}(x)$$

- The activation at layer i is given by

$$h_i(x) = g(a_i(x))$$

- g is an element wise function and is called the **activation function**.
- The activation at the output layer is given by

$$f(x) = h_L(x) = O(a_L(x))$$

  where O is the output activation function
- For ease of notation $a_i(x) = a_i$ and $h_i(x) = h_i$.
- We also have $h_0 = x$ and $h_L = \hat{y} = \hat{f}(x)$
- We can write $\hat{y}$ as

$$\hat{y}_i = \hat{f}(x_i) = O(W_3 g(W_2 g(W_1 x_i + b_1) + b_2) + b_3)$$

  This becomes our model as specified in 3.2.
- Parameters become $\theta = W_1, ..., W_L, b_1, ..., b_L$
- We can now write our gradient descent algorithm more concisely as

Algorithm 4 Gradient Descent Modified

```
    GRADIENT-DESCENT( )
1:  t ← 0;
2:  max_iterations ← 1000;
3:  Initialize θ_0 = [W_1^0, ..., W_L^0, b_1^0, ...., b_L^0]
4:  while t++ < max_iterations do
5:      θ_{t+1} ← θ_t − η∇θ_t
6:  end while
```

where we have

$$\theta = [W_1, ..., W_L, b_1, ..., b_L] \text{ and } \nabla\theta_t = [\frac{\partial \mathcal{L}(\theta)}{\partial W_{1,t}}, ..., \frac{\partial \mathcal{L}(\theta)}{\partial W_{L,t}}, \frac{\partial \mathcal{L}(\theta)}{\partial b_{1t}}, ..., \frac{\partial \mathcal{L}(\theta)}{\partial b_{1t}}]^T$$

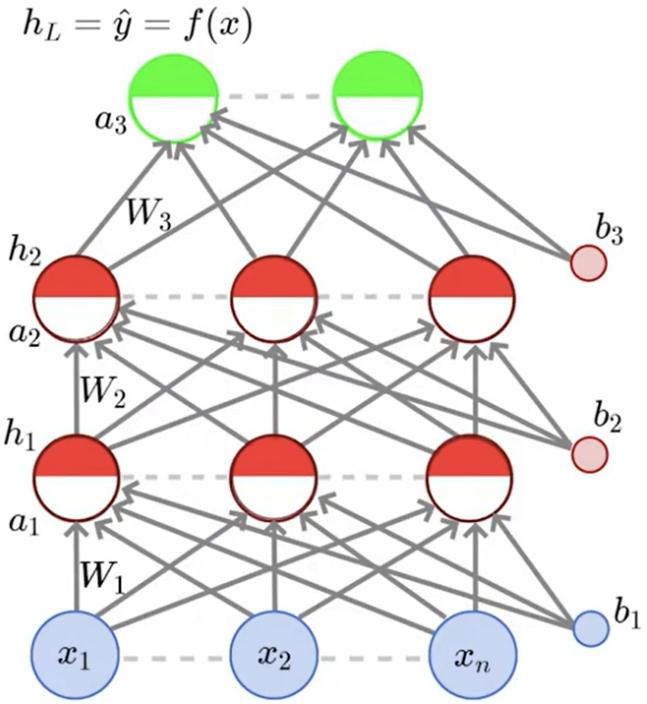


Figure 6: Feed Forward Network

4.2 Output functions and Loss functions

- Assume we are trying to predict values in the range of all real values, an appropriate loss function would be mean squared error function.

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k}(\hat{y}_{ij} - y_{ij})^2$$

- If we want to predict values in the range of real values, regression, then we can take the output function as a linear function

$$f(x) = h_L = O(a_L) = W_O a_L + b_O$$

- Given that we are performing classification, then we can use cross entropy loss function. This is known as negative log likelihood function.

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k}(y_{ij}log(\hat{y}_{ij}) + (1 - y_{ij})log(1 - \hat{y}_{ij}))$$

- Ensure that $\hat{y}$ is a probability distribution, one appropriate output function can be the softmax function

$$\hat{y}_j = O(a_L)_j = \frac{e^{a_{L,j}}}{\sum_{i=1}^{k} w^{a_{L,i}}}$$

	Real Values	Probabilities
Output Activation	Linear	Softmax
Loss Function	Squared Error	Cross Entropy

Table 4: Choice of Output and Loss functions based on type of Output

4.3 Backpropagation

- Intuitively it can be written as

$$\underbrace{\frac{\partial \mathscr{L}(\theta)}{\partial W_{111}}}_{\text{Talk to the weight directly}} = \underbrace{\frac{\partial \mathscr{L}(\theta)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_3}}_{\text{Talk to the output layer}} \underbrace{\frac{\partial a_3}{\partial h_2} \frac{\partial h_2}{\partial a_2}}_{\text{Talk to the previous hidden layer}} \underbrace{\frac{\partial a_2}{\partial h_1} \frac{\partial h_1}{\partial a_1}}_{\text{Talk to the previous hidden layer}} \underbrace{\frac{\partial a_1}{\partial W_{111}}}_{\text{and now talk to the weights}}$$

Figure 7: Backpropagation Intuition

- For further discussion, the output function is considered to be softmax and the loss function is considered to be cross entropy.
- **Gradient w.r.t output layer**: We start with the part "Talk to the output layer".

$$\mathcal{L}(\theta) = -\log \hat{y}_l \ (l = \text{ true class label})$$

$$\frac{\partial}{\partial \hat{y}_i}\mathcal{L}(\theta) = \begin{cases} -\frac{1}{\hat{y}_l}, & \text{if } i = l \\ 0, & \text{otherwise} \end{cases} = [0, ..., 0, -\frac{1}{\hat{y}_l}, 0, ..., 0]^T = -\frac{1}{\hat{y}_l} e_l$$

$e_l$is a $k-$dimensional vector whose l^{th} element is 1 and rest are 0.

$$\frac{\partial \mathcal{L}(\theta)}{\partial a_{L,i}} = -\frac{1}{\hat{y}_l} \frac{\partial \hat{y}_l}{\partial a_{L,i}}$$

Since, $\hat{y}_l$ is calculated using cross entropy it is dependent on all a_L, i

$$\hat{y}_j = \frac{e^{a_{Lj}}}{\sum_i e^{a_{Li}}}$$

$$\frac{\partial \hat{y}_l}{\partial a_{L,i}} = \begin{cases} \hat{y}_l(1 - \hat{y}_l), & \text{if } i = l \\ -\hat{y}_l \hat{y}_i, & \text{if } i \neq l \end{cases}$$

We can now write the entire derivative in one shot as

$$\frac{\partial \mathcal{L}(\theta)}{\partial a_{L,i}} = -\frac{1}{\hat{y}_l}(\hat{y}_l)(1_{l=i} - \hat{y}_i) = -(e_l - \hat{y})$$

- **Chain rule among multiple paths**: If a function $p(z)$ can be written as a function of intermediate results $q_i(z)$ then we have

$$\frac{\partial p(z)}{\partial z} = \sum_m \frac{\partial p(z)}{\partial q_m} \frac{\partial q_m}{\partial z}$$

- **Gradient w.r.t hidden units**: Now we "Talk to the hidden layers", in this case $p(z)$ is the loss function, $z = h_{ij}$ and $q_m(z) = a_{Lm}$. We have

$$\frac{\partial \mathcal{L}(\theta)}{\partial h_{ij}} = \sum_{m=1}^{k} \frac{\mathcal{L}(\theta)}{\partial a_{i+1,m}} W_{i+1,m,j}$$

$$\frac{\partial \mathcal{L}(\theta)}{\partial a_{ij}} = \frac{\partial \mathcal{L}(\theta)}{\partial h_{ij}} \frac{\partial h_{ij}}{\partial a_{ij}} = \frac{\partial \mathcal{L}(\theta)}{\partial h_{ij}} g'(a_{ij})$$

- **Gradient w.r.t Parameters**: Finally, we "talk to the weights"

$$\frac{\partial \mathcal{L}(\theta)}{\partial W_{kij}} = \frac{\partial \mathcal{L}(\theta)}{\partial a_{ki}} \frac{\partial a_{ki}}{\partial W_{kij}} = \frac{\partial \mathcal{L}(\theta)}{\partial a_{ki}} h^T_{k-1,j}$$

$$\frac{\partial \mathcal{L}(\theta)}{\partial b_{ki}} = \frac{\partial \mathcal{L}(\theta)}{\partial a_{ki}} \frac{\partial a_{ki}}{\partial b_{ki}} = \frac{\partial \mathcal{L}(\theta)}{\partial a_{ki}}$$

- We can now write the full pseudocode as

Algorithm 5 Forward Propagation

for $k = 1$ to $L - 1$ **do**
  $a_k = b_k + W_k h_{k-1}$
  $h_k = g(a_k)$
end for
$a_L = b_L + W_L h_{L-1}$
$\hat{y} = O(a_L)$

Algorithm 6 Backward Propagation

$\nabla_{a_L} \mathcal{L}(\theta) = -(e(y) - \hat{y})$
for $k = L - 1$ to 1 **do**
  $\nabla_{W_k} \mathcal{L}(\theta) = \nabla_{a_k} \mathcal{L}(\theta) h_{k-1}^T$
  $\nabla_{b_k} \mathcal{L}(\theta) = \nabla_{a_k} \mathcal{L}(\theta)$
  $\nabla_{h_{k-1}} \mathcal{L}(\theta) = W_k^T \nabla_{a_k} \mathcal{L}(\theta)$
  $\nabla_{a_{k-1}} \mathcal{L}(\theta) = \nabla_{h_{k-1}} \mathcal{L}(\theta) \odot [..., g'(a_{k-1,j}), ...]$
end for

- g' for logistic function $\sigma(z)$ is $g(z)(1 - g(z))$ and for tanh function is $1 - (g(z))^2$.
- On flat surfaces, gradient descent moves very slow. How do we solve this?

5 Gradient Descent Types

5.1 Momentum based Gradient Descent

- **Intuition**: If I am repeatedly being asked to move in the same direction then I should probably gain some confidence and start taking bigger steps in that direction. Just as a ball gains momentum while rolling down a slope.
- Update rule for momentum based gradient descent is

$$u_t = \beta u_{t-1} + \nabla w_t, \qquad u_0 = \nabla w_0,\ u_{-1} = 0$$
$$w_{t+1} = w_t - \eta u_t$$

- We are not only considering the current gradient, but we are also giving some importance to past history. β is typically less than 1, so we give decreasing importance to previous histories. We have

$$u_t = \beta u_{t-1} + \nabla w_t = \beta^2 u_{t-2} + \beta \nabla w_{t-1} + \nabla w_t = ... = \sum_{\tau=0}^{t} \beta^{t-\tau} \nabla w_\tau$$

- In addition to current update, also look at the history of updates.
- Even in the regions having gentle slopes, momentum based gradient descent is able to take large steps because the momentum carries it along.
- However, there is a possibility of overshooting our goal. This overshoot could lead to oscillations as well.
- Despite these oscillations, it will converge faster than gradient descent.

5.2 Nesterov Accelerated Gradient Descent

- Can we do something to reduce the oscillations observed in momentum based gradient descent?
- **Intuition**: Look before you leap, we ask the weights to move by two parts βu_{t-1} and ∇w_t. The idea is to first move by βu_{t-1} and then compute ∇w_t. So instead of relying only on current gradient we are essentially "looking ahead" and computing new gradient.
- Update rule for NAG is

$$u_t = \beta u_{t-1} + \nabla(w_t - \beta u_{t-1})$$
$$w_{t+1} = w_t - \eta u_t$$
$$\text{with } u_{-1} = 0, \text{ and } 0 \leq \beta \leq 1$$

5.3 Stochastic vs Batch Gradient Descent

- Regular gradient descent goes over the entire data once before updating the parameters. Because this is the true gradient of the loss as derived earlier. Hence, theoretical guarantees hold.
- Imagine we have a million points in the training data, then it will take very long.
- In **stochastic** version, we update the parameters for every point in the data. Now if we have a million data points then we will make a million updates in each epoch.
- However, this is an approximate gradient. Hence, we have no guarantee that each step will decrease the loss.
- So, going over a large data once is bad, going over a single point and updating is bad, but going over some data points and updating is ok.
- This is the idea for **mini-batch** gradient descent.
- 1 epoch is one pass over the entire data, 1 step is one update of the parameters, N is the number of data points, and B is the mini batch size.
- So, for regular gradient descent the number of epochs is the same as number of steps, for stochastic gradient descent the number of steps is the same as N, and for mini-batch gradient descent the number of steps is the same as $\frac{N}{B}$.
- It is intuitive that a larger batch size is better, but we sacrifice on time.

Algorithm	**# of steps in 1 epoch**
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini Batch Gradient Descent	$\frac{N}{B}$

Table 5: Stochastic vs Mini Batch Gradient Descent

5.4 Scheduling Learning Rate

- Instead of using momentum and NAG we could have simply increased the learning rate, but on regions which have a steep slope, the already large gradient would blow up farther.
- What we need is for the learning rate to be small when gradient is high and vice versa.
- Tune learning rate, try different values on a log scale: $0.0001, 0.001, 0.001, 0.1, 1.0$.
- Run a few epochs with each of these and figure out a learning rate which works best. Now do a finer search around this value.
- These are just heuristics, no clear winning strategy.
- **Annealing learning rate**: Decrease the learning rate as we get close to the minima.
- **Step Decay**: Halve the learning rate after every 5 epochs or halve the learning rate after an epoch if the validation error is more than what it was at the end of the previous epoch.
- **Exponential Decay**: $\eta = \eta_0^{-kt}$, where η_0 and k are hyperparameters and t is a step number. However, choosing k becomes complex.
- **1/t Decay**: $\eta = \frac{\eta_0}{1+kt}$, again choosing k is a bit tricky, ideally we won't use these learning rates.
- For momentum the following schedule was suggested by Sutskever et al., 2013

  $$\beta_t = min(1 - 2^{-1-\log_2(\lfloor \frac{t}{250} \rfloor + 1)}, \beta_{max})$$

  β_{max} is chosen from $\{0.999, 0.995, 0.99, 0.9, 0\}$
- In practice, often a line search is done to find a relatively better value of η. Update w using different values of η then pick the best w based on loss function.

6 Adaptive Learning Rates

- **Intuition**: Decay the learning rate for parameters in proportion to their update history(more updates means more decay).
- Update rule for **AdaGrad**

$$v_t = v_{t-1} + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \times \nabla w_t$$

  and similar set of equations for b_t.

- **Intuition**: AdaGrad decays the learning rate very aggressively(as the denominator grows). As a result, after a while, the frequent parameters will start receiving very small updates. To avoid this we can decay the denominator.
- Update rule for **RMSprop**

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \times \nabla w_t$$

  Give lower and lower weightage to previous gradients

- RMSprop converges more quickly than AdaGrad by being less aggressive on decay.
- In RMSprop, the deltas start oscillating after a while, learning rate can increase, decrease or remain constant. Whereas in AdaGrad learning rate can only decrease as the denominator constantly grows.
- In case of oscillations, setting η_0 properly could solve the problem.
- Both are **sensitive** to initial learning rate, initial conditions of parameters and corresponding gradients.
- **AdaDelta** avoids setting initial learning rate η_0

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$$
$$\Delta w_t = -\frac{\sqrt{u_{t-1} + \epsilon}}{\sqrt{v_t + \epsilon}} \nabla w_t$$
$$w_{t+1} = w_t + \Delta w_t$$
$$u_t = \beta u_{t-1} + (1-\beta)(\Delta w_t)^2$$

  Now the numerator, in the effective learning rate, is a function of past gradients. Observe that the u_t that we compute at t will be used only in the next iteration. Essentially one history runs behind the other.

- After some i iterations, v_t will start decreasing and the ratio of the numerator to the denominator starts increasing. If the gradient remains low for a subsequent time steps, then the learning rate grows accordingly.
- Therefore, AdaDelta allows the numerator to increase or decrease based on the current and past gradients.
- **Intuition**: Do everything that RMSprop does to solve the decay problem of AdaGrad, plus used a cumulative history of the gradients.
- Update rule for **Adam(Adaptive Moments)**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla w,\ \hat{m}_t = \frac{m_t}{1-\beta_1^t} \leftarrow \text{Incorporating classical momentum}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2,\ \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

  Note that we are taking a running average of the gradients as m_t.

- One way of looking at this is that we are interested in the expected value of the gradient and not on a single point estimate computed at time t.
- $\hat{m}_t$ and $\hat{v}_t$ are bias correction terms, we do this to avoid very high learning rate update. It can also be derived by taking expectation of m_t.

- Bias correction is the problem of exponential averaging.
- General form of L^p norm is $(|x_1|^p + |x_2|^p + ... + |x_n|^p)^{\frac{1}{p}}$, as $p \to \infty$, L^p boils down to taking the max value.
- In Adam, we called $\sqrt{v_t}$ as L^2 norm, why not replace it with L^∞ norm. Therefore, we have

$$v_t = \max(\beta_2^{t-1}|\nabla w_1|, \beta_2^{t-2}|\nabla w_2|, ..., |\nabla w_t|)$$
$$v_t = \max(\beta_2 v_{t-1}, |\nabla w_t|)$$
$$w_{t+1} = w_t - \frac{\eta_0}{v_t}\nabla w_t$$

  Observe that we didn't use bias corrected v_t as max norm is not susceptible to initial zero bias.
- Suppose that we initialize w_0 such that the gradient at the w_0 is high. Suppose further that the gradients for the next subsequent iterations are also zero because x is sparse. Ideally, we don't want the learning rate to change its value when $\nabla w_t = 0$.
- Update Rule for **MaxProp** is similar to RMSprop

$$v_t = \max(\beta v_{t-1}, |\nabla w_t|)$$
$$w_{t+1} = w_t - \frac{\eta}{v_t + \epsilon}\nabla w_t$$

- We can extend the same idea to Adam and call it **AdaMax**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla e_t,\ \hat{m}_6 = \frac{m_t}{1-\beta_1^t}$$
$$v_t = \max(\beta_2 v_{t-1}, |\nabla w_t|)$$
$$w_{t+1} = w_t - \frac{\eta}{v_t + \epsilon}\hat{m}_t$$

- **Intuition**: We know NAG is better than momentum based GD, why not just incorporate it with Adam.
- We can rewrite NAG such that there is no $t-1$ term, as

$$g_{t+1} = \nabla w_t$$
$$m_{t+1} = \beta m_t + \eta g_{t+1}$$
$$w_{t+1} = w_t - \eta(\beta m_{t+1} + g_{t+1})$$

- Update rule for **NAdam(Nesterov Adam)**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla w,\ \hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2,\ \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon}(\hat{m}_{t+1} + \frac{(1-\beta_1)\nabla w_t}{1-\beta_1^{t+1}})$$

- It is common for new learners to start out using off the shelf optimizers, which are later replaced by custom designed ones.
- **Cyclical Learning Rate**: Suppose the loss surface has a saddle point. Suppose further that the parameters are initialized, and the learning rate is decreased exponentially. After some iterations, the parameters will reach near the saddle point. Since, the learning rate has decreased exponentially, the algorithm has no way of coming out of the saddle point.
  What if we allow the learning rate to increase after some iterations.
- **Rationale**: Often, difficulty in minimizing the loss arises from saddle points rather than poor local minima.
- A simple way to solve this is to vary the learning rate cyclically. One example is triangular learning rate.
- **Cosine Annealing(Warm Re-Start)**: also based on cyclical learning rate

$$\eta_t = \eta_{min} + \frac{\eta_{max} - \eta_{min}}{2}(1 + \cos(\pi\frac{t\%(T+1)}{T})),\ T \text{ is restart interval}$$

- **Warm-start**: Using a low initial learning rate helps the model to warm and converge better.

7 Regularization

7.1 Introduction to Bias and Variance

- Deep learning models typically have BILLIONS of parameters whereas the training data may have only MILLIONS of samples. Therefore, they are called **over-parameterized** models.
- Over-parameterized models are **prone** to a phenomenon called **over-fitting**.
- Simple models trained on different samples of the data do not differ much from each other. However, they are very far from the true sinusoidal curve (**under fitting**).
- **Bias**: Difference between the expected value of predictions subtracted by the true value.

$$\text{Bias } \hat{f}(x) = E[\hat{f}(x)] - f(x)$$

  Simple models will have high bias and complex models will have low bias.

- Variance is defined as

$$\text{Variance } \hat{f}(x) = E[(\hat{f}(x) - E[\hat{f}(x)])^2] = E[\hat{f}^2(x)] - E[\hat{f}(x)]^2$$

  Roughly speaking it tells us how much the different $\hat{f}(x)'s$ trained on different samples of the data differ from each other. Simple models have a low variance whereas complex model have a high variance.

	Bias	Variance
Simple Model	High	Low
Complex Model	Low	High

- There is always a trade-off between the bias and variance. Both contribute to the MSE loss.
- Consider a new point (x, y) which was not seen during training. If we use the model $\hat{f}(x)$ to predict the value of y then the mean squared error is given by $E[(y - \hat{f}(x))^2]$ (MSE), we can show that

$$E[(y - \hat{f}(x))^2] = \text{ Bias}^2 + \text{ Variance } + \sigma^2 \text{ (irreducible error)}$$

- The parameters of $\hat{f}(x)$ are trained using a training set. However, at test time we are interested in evaluating the model on a validation(unseen) set which was not used for training. This gives rise to two entities $train_{err}$ and $test_{err}$. Typically, this gives rise to the following graph, parabola represents test error.

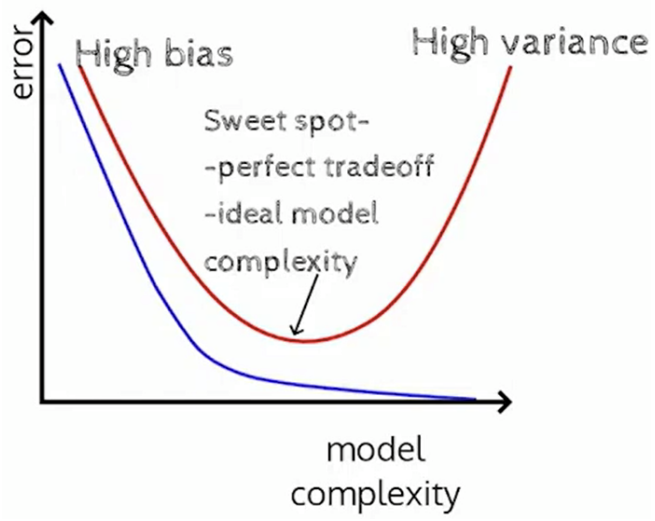


Figure 8: Train vs Test Error

- Complex model is more sensitive to minor changes in the data, as compared to simple models. Hence while training, instead of minimizing the training error we should minimize train error $+\Omega(\theta)$, which is high for complex models and small for simple models.
- This is the basis for all the regularization methods.

7.2 Regularization Methods

- For **L2 Regularization** define the loss as

$$\bar{L}(w) = L(w) + \frac{\alpha}{2}\|w\|^2$$

  and the following gradient would look like

$$\nabla\bar{L}(w) = \nabla L(w) + \alpha w$$

  So it is very easy to implement this.

- It essentially makes it, so the weights are closer to 0, non-essential weights will be closer to 0 as compared to essential weights.
- **Data Augmentation**: Consider images, rotate them, shift them, blur them, change some pixels, etc.
- We exploit the fact that certain transformations to the image do not change the label of the image.
- Typically, more data = better learning.
- Augmentation works well for image classification/object recognition tasks.
- It has been shown, that it works well for speech as well.
- **Parameter Sharing** is used in CNNs, essentially same filter is applied at different positions of the image, or same weight matrix acts on different input neurons.
- This is typically used in autoencoders, encoder and decoders.
- **Injecting Noise at input**: We can show that for a simple input output neural network, adding Gaussian noise to the input is equivalent to weight decay(L2 regularization).
- We essentially shift the input by a slight amount at every epoch, so the model sees a different input at every epoch making it harder to over fit.
- This can also be viewed as data augmentation.
- We can similarly inject noise at output as well.
- **Early stopping**: Have a patience parameter p, after p steps if validation error does not decrease then we stop and return the model p steps before.
- This is very effective and widely used, it can be used even with other regularizers.
- **Ensemble Methods**: Combine the output of different models to reduce generalization error. These models can correspond to different classifiers, it could be different instances of the same classifier trained with different hyperparameters, features, samples etc.
- **Bagging**: Form an ensemble using different instances of the same classifier. For a given dataset, construct multiple training sets by sampling with replacement.
- Bagging would work well when the errors of the model are independent or uncorrelated, if they are correlated then the mse of the ensemble is as bad as the individual model.
- On average, the ensemble will perform at least as well as its individual members.
- Training several neural networks for making an ensemble is prohibitively expensive.
- **Dropout**: Refers to dropping out units, temporarily remove a node and all its incoming and outgoing connections resulting in a thinned network.
- Each node is retained with a fixed probability, $p = 0.5$, for hidden nodes and $p = 0.8$ for visible nodes.
- We initialize all the parameters of the network and start training. For the first training instance(or mini-batch), we apply dropout resulting in the thinned network. Then update only those parameters that are active.
- For the second training instance, we again apply dropout resulting in a different thinned network. We again compute the loss and back propagate.
- Each thinned network gets rarely trained but the parameter sharing ensures that no model has untrained or poorly trained parameters.
- Dropout essentially applies a masking noise to the hidden units. Prevents hidden units from coadapting.

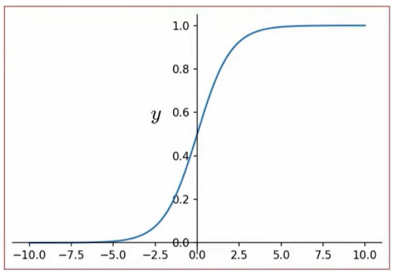
8 Deep Learning Revival

8.1 Unsupervised Pre-Training

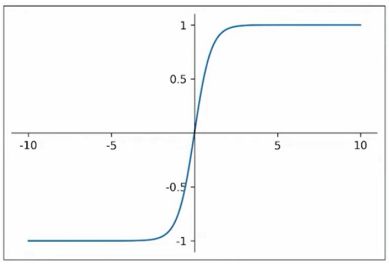
- We will first train the weights between the layers using an **unsupervised objective**, essentially we will try to reconstruct x.
- If we are able to do this, it would mean that the hidden layers are capturing all the important information of the image. We will be learning one layer at a time.
- h_1 will try to reconstruct x, h_2 will try to reconstruct h_1, and so on...
- After this layerwise pre-training, we add the output layer and train the whole network.
- In effect we have initialized the weights of the network using the greedy unsupervised objective and are now fine-tuning these weights using the supervised objective.
- This helps in optimization and regularization.
- Unsupervised objective ensures that the learning is not greedy w.r.t the supervised objective.
- Some other experiments have also shown that pre-training is more robust to random initializations.
- This led to people thinking that deep networks are sensitive to initial weights, and maybe if we have better initial weights it would lead to better network.

8.2 Better Activation Functions

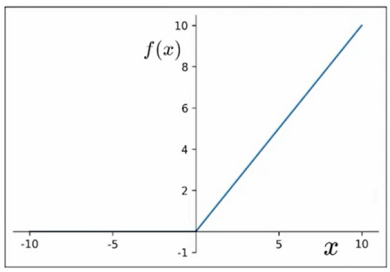
- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$, compresses all its inputs to the range $[0, 1]$. $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$, once sigmoid neuron is **saturated**($\sigma(x) = 0$ or 1) then the training will halt as the derivative would become zero.
  Another issue is that sigmoid are **not zero centered**, leading to all positive or all negative gradients, lack of diversity. These are also **computationally expensive** because of the exponential term.
- **tanh**: $tanh(x)$, compresses all its input to the range $[-1, 1]$ so, its **zero centered**.$\frac{\partial tanh(x)}{\partial x} = (1 - tanh^2(x))$, but we still face the issue of **vanishing gradient**, and it is still **computationally expensive**.
- **Rectified Linear Unit**: $ReLU(x) = \max(0, x)$, does not saturate in the positive region, it is computationally efficient and in practice converges much faster than sigmoid and tanh.
  Clearly the derivative is 1 for $x > 0$ and 0 otherwise, leading to no updates for negative neurons, which makes the neuron dead.
  In practice a large fraction of ReLU units can die if the learning rate is set too high.
  It is advised to initialize the bias to a positive value.
- **Leaky ReLU**: $f(x) = \max(0.1x, x)$, no saturation, computationally efficient, close to zero centered outputs, and will not die.
- **Parametric ReLU**: $f(x) = \max(\alpha x, x)$, α can be learned.
- **Exponential Linear Unit**: $ELU = \begin{cases} x \text{ if } x > 0 \\ ae^x - 1 \text{ if } x \leq 0 \end{cases}$, close to zero centered outputs, expensive.



(a) Sigmoid Function

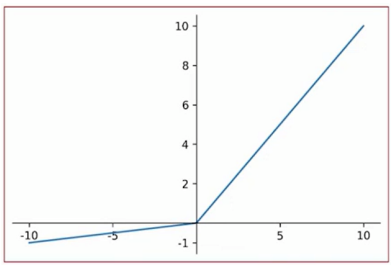


(b) Tanh Function

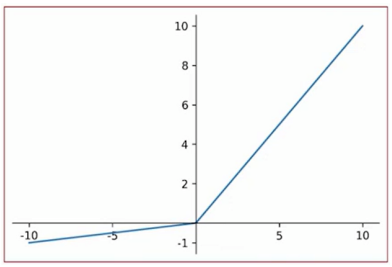


(c) ReLU Function

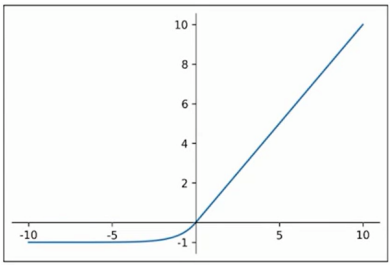
Figure 9: Activation Functions - 1



(a) Leaky ReLU Function



(b) Parametric ReLU Function



(c) ELU Function

Figure 10: Activation Functions - 2

- **Model Averaging**: Train k models with independent weights$(w_1, w_2, ..., w_k)$ on independently sampled data points from the original dataset. Each model is expected to be good at predicting a subset of training samples well.
  During inference, the prediction is done by averaging the predictions from all the models, it is often **arithmetic or geometric** mean for regression problems.
- In bagging, the subset of training samples seen by each of the models does not change across epochs. Therefore, the model weights get optimized for those samples.
- Each sub-model sees only some parts of the training data, therefore we want updates to be larger.
- We don't have this luxury in case of dropout.
- **MaxOut**: In a way is a variation of Dropout, but instead of having probability, we take say 3 neurons in a layer and propagate the max of them. Can also be thought of as a generalization of ReLU and Leaky ReLU. Two MaxOut neurons with sufficient number of affine transformations, act as a universal approximator.
- **Gaussian Error Linear Unit**: $GELU = mx$ where $m \sim Bernoulli(\Phi(x))$
- **Swish**: $x\sigma(\beta x)$, taking $\beta = 1.702$ we get GELU, and taking 1 we get SILU(Sigmoid-weighted Linear Unit).

9 Convolutional Neural Networks

- Convolution operation can be defined as

$$x * w = \sum_{a=0}^{\infty} x(t-a)w(a) = \sum_{a=0}^{\infty} x(a)w(t-a)$$

  where w is called a filter and x is the input.
- Similarly 2D convolution operation can be defined as

$$(I * K)(i,j) = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I(i-a, j-a)K(a,b)$$

  We can use this for images.
- We can define correlation operator as

$$g(i,j) = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I(i+a, j+a)K(a,b)$$

  Correlation and Convolution operator will be same when K is a symmetric matrix.
- For practical implementations most libraries implement correlation operator rather than convolution.

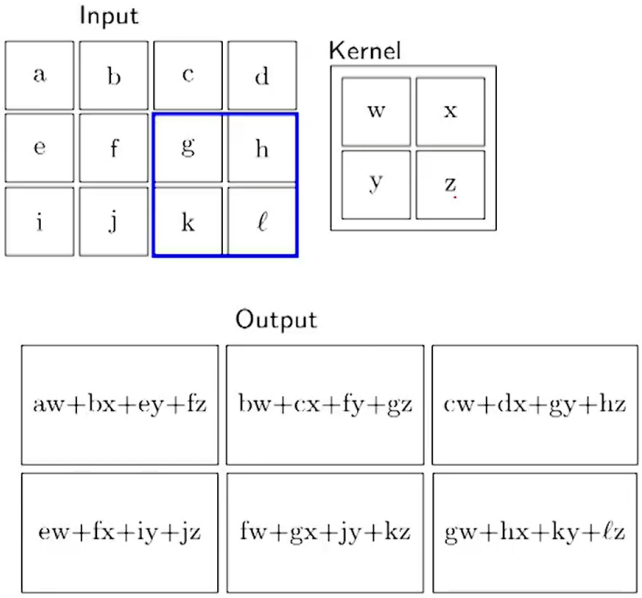


Figure 11: Convolution Example

- When we convolve a filter with an image the resulting image is called a feature map.
- If we use multiple such filters we will get multiple feature maps, and then we stack them, this number will in a way act as the number of color channels.
- Let image be of size $W_1 \times H_1 \times D_1$, let the number of filters be K, let the output be of size $W_2 \times H_2 \times D_2$, and let stride be S and spatial extent F. The filters are of size $F \times F \times D_1$.

$$W_2 = 1 + \frac{W_1 - F}{S} \quad H_2 = 1 + \frac{H_1 - F}{S} \quad D_2 = K$$

- If we want the output to be of the same size as input, we add padding.

$$W_2 = 1 + \frac{W_1 - F + 2P}{S} \quad H_2 = 1 + \frac{H_1 - F + 2P}{S} \quad D_2 = K$$

- One important characteristic of CNNs is **weight sharing**.
- **Pooling**: Basically the same as a filter, except instead of multiplying with numbers, we take the max/min/average of the pixel intensities within the filter window.
- Pooling essentially helps lower the size of the feature maps, and extract only essential or important features.
- A simple CNN will have (Convolution→Pooling)×2 layers, then a normal ANN.
- Pooling layers will propagate gradient in backpropagation, as they do not have any weight attached to them.
- For Max pooling, the gradient from the next layer is propagated only to the location of the maximum value in the pooling window.
- For Average pooling, the gradient from the next layer is distributed equally to all elements in the pooling region because each element contributes equally to the average.
- For convolution, the backpropagation update would be convolving the input(X) to the convolution layer with the loss or gradient values given by the pooling layer.
- And when we want to propagate the gradient we will convolve weights(W) instead of X.

9.1 Different Architectures

- **AlexNet**: Input size is of $227 \times 227 \times 3$, the architecture can be seen below. Approx $27M$ parameters.

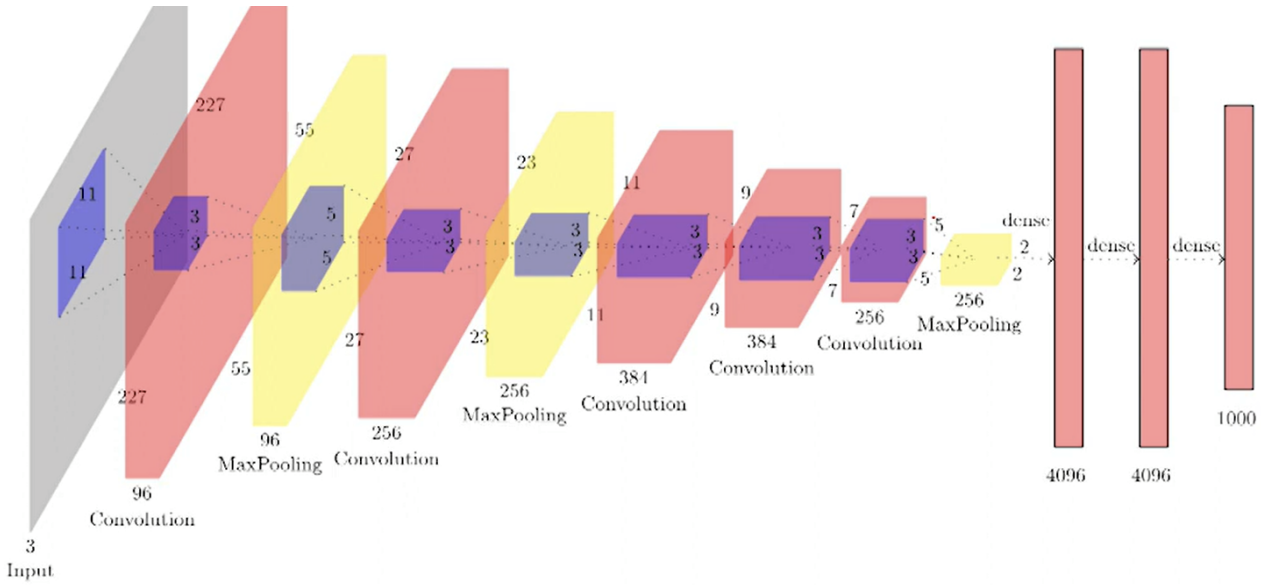


Figure 12: AlexNet Architecture

- **ZFNet**: Input size is the same, the architecture is basically the same as AlexNet except the three consecutive convolution layers have filters $512, 1024, 512$ respectively.
- **VGGNet**: Input is again the same, the key idea in this is kernel size is always 3×3.
- **GoogleLeNet**: Key idea is to have multiple sized filters to capture features at varying range.

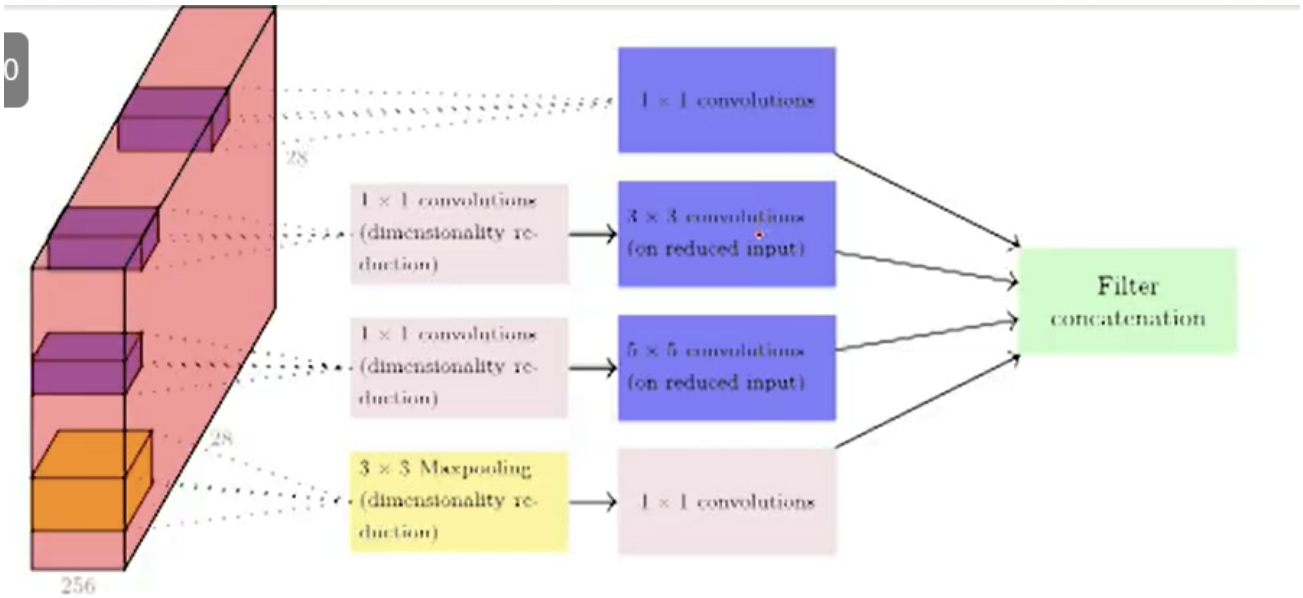


Figure 13: Inception Model

- **ResNet**: Train a shallow network, then add layers and train deeper network. We can ensure this by adding a skip connection.

10 Word Representations

10.1 Direct Representations

- **Corpus**: Group of documents.
- Consider the set V of all uniques words across all input streams, V is called the **vocabulary** of the corpus. We need a representation for every word in V.
- We will be using the following corpus, to understand each representation
  1. Human machine interface for computer applications
  2. User opinion of computer system response time
  3. User interface management system
  4. System engineering for improved response time
- Vocabulary for this corpus is V = [human, machine, interface, for, computer, applications, user, opinion, of, system, response, time, interface, management, engineering, improved]
- One very simple way of doing this is to use **one-hot vectors** of size $|V|$, essentially switch the bit on if the word is present else it is off.
- **machine** would be represented as $\begin{bmatrix}0 & 1 & 0 & ... & 0 & 0\end{bmatrix}$
- V tends to be very large, 50K for PTB, 13M for Google IT corpus.
- These representations do not capture any notion of similarity.
- With one hot representations, the Euclidean distance between any two words is $\sqrt{2}$ and the cosine similarity between any two words is 0.
- We want representations of cat and dog to be closer to each other than the representations of cat and truck.
- These are also called **sparse representations**.
- A co-ocurrence matrix is a terms $\times$ terms matrix which captures the number of times a term appears in the context of another term
- The context is defined as a window of k words around the terms.
- With $k = 2$, the co-ocurrence matrix for the above corpus would be

	human	machine	system	for	...	user
human	0	1	0	1	...	0
machine	1	0	0	1	...	0
system	0	0	0	1	...	2
for	1	1	1	0	...	0
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
user	0	0	2	0	...	0

Table 6: Co-Ocurrence Matrix

- Essentially for the word w_1, we check a window of k words around it and count how many times other words appear.
- This is also known as a word $\times$ context matrix.
- We could choose the set of **words** and **contexts** to be same or different.
- Each row (column) of the co-ocurrence matrix gives a representation of the corresponding word (context).
- Stop words(a, the, for, etc.) are very frequent $\rightarrow$ these counts will be very high.
- We can solve this by ignoring very frequent words or use a threshold t, and if a word, context pair goes above a threshold then we cap it.

- Instead of count we can also use PMI

$$PMI(w,c) = \log \frac{p(c|w)}{p(c)} = \log \frac{count(w,c) \cdot N}{count(c) \cdot count(w)}$$

  Obviously, when $count(w,c) = 0$ then PMI becomes $-\infty$, which is an issue.

- We can solve this by taking PMI value as 0 when $count(w,c) \leq 0$ and using the formula when $count(w,c) > 0$, this is called $PMI_0(w,c)$.

- Another solution is instead of $count(w,c) > 0$, we replace with $PMI(w,c) > 0$. This is called Positive PMI or $PPMI$.

	human	machine	system	for	...	user
human	0	2.944	0	2.25	...	0
machine	2.944	0	0	2.25	...	0
system	0	0	0	1.15	...	1.84
for	2.25	2.25	1.15	0	...	0
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
user	0	0	1.84	0	...	0

Table 7: PMI Co-Ocurrence Matrix

- This is however still very large and very sparse, and it grows with vocabulary.

- Consider $X = X_{PPMI_{m \times n}}$, we can use singular value decomposition to get a rank k approximation.

$$\hat{X}_{m \times n} = U_{m \times k} \Sigma_{k \times k} V^T_{k \times n}$$

- SVD gives the best rank k approximation of the original data, Discovers latent semantics in the corpus.

- SVD will essentially extract the most important features from the matrix.

- Conventionally, we take word representation as

$$W_{word} = U\Sigma \in \mathbb{R}^{m \times k}$$

  This is representation of the m words in the vocabulary and $W_{context} = V$ is taken as the representation of the context words.

- This idea comes from the fact that $\hat{X}\hat{X}^T$ and $W_{word}W^T_{word}$ are the same matrices, i.e., they roughly capture the same cosine similarity.

- These methods are called count based models because they use the co-ocurrence counts of words.

10.2 Bag of Words Model

- **Task**: Predict the nth word given previous $n-1$ words.

- **Training data**: All n word windows in the corpus.

- Consider doing this for $n = 2$, predict the second word given the first.

- We can model this problem using a feedforward neural network with one hidden layer. This network will take input a one-hot representation of the context word and output a probability vector of words.

- Parameters are $W_{context} \in \mathbb{R}^{k \times |V|}$ and $W_{word} \in \mathbb{R}^{|V| \times k}$, and we use softmax to get probabilities.

- The product $W_{context}x$ is simply the ith column of $W_{context}$, give x is a one hot vector.

- When the i^{th} word is present the i^{th} element in the one hot vector is ON and the i^{th} column of $W_{context}$ gets selected.

- There is a one to one correspondence between the words and columns of $W_{context}$.

- We can treat the i^{th} column of $W_{context}$ as the representation of the context i.

- $P(word = i|v)$ depends on the i^{th} column of W_{word}.
- We thus treat the i^{th} column of W_{word} as the representation of the word i.
- Consider the context word with index c and the correct output with index w

$$L(\theta) = -\log \hat{y}_w = -\log P(w|c)$$
$$h = W_{context} x_c = u_c$$
$$\hat{y}_w = \frac{e^{u_c v_w}}{\sum_{w' \in V} e^{u_c v_{w'}}}$$

  u_c is the column of $W_{context}$ corresponding to context word c and v_w is the column of W_{word} corresponding to the word w.
- Give an context u_c if we are predicting v_w then the update rule for v_w derived from backpropagation is

$$v_w = v_w + \eta u_c (1 - \hat{y}_w)$$

  This increases the cosine similarity between u_c and v_w.
- The training objective ensures that the cosine similarity between word (v_w) and context word (u_c) is maximized
- In practice, instead of window size of 1 it is common to use a window size of d.
- We can simply concatenate all the one-hot representations, and the first layer weights changes to $[W_{context}, W_{context}, ...d \text{ times}]$, this becomes very complex. Instead, we simply add the column where bit is on.

$$h = \sum_{i=1}^{d-1} u_c, \text{ where } u_c \text{ is the column of } W_{context} \text{ where bit was on}$$

- Computation of softmax is expensive as denominator requires sum of all words in vocabulary.

10.3 Skip Gram Model

- Predicts context words given an input word.
- the role of context and word has changed now.
- It essentially becomes the same as bag of words, and we run into the same problems.
- **Negative Sampling**
- Let D be the set of all correct (w, c) pairs in the corpus
- Let D' be the set of all incorrect (w, r) pairs in the corpus
- D' can be constructed by randomly sampling a context word r which has never appeared with w and creating a pair (w, r)
- We are interested in maximizing

$$\prod_{(w,c) \in D} P(z = 1|w, c) \prod_{(w,r) \in D'} P(z = 0|w, r) = \sum_{(w,c) \in D} \log \sigma(v_c^T v_w) + \sum_{(w,r) \in D'} \log \sigma(-v_r^T v_w)$$

- The size of D' is k times the size of D.
- **Contrast Estimation**: Consider instead of outputting probability we output a score.
- Now, we want the score of correct word s to be higher than wrong word s_r, so we try to maximize $s - s_r$.
- But we would like the difference to be at least m, so we maximize $s - (s_r + m)$
- If $s > s_r + m$ then we don't do anything.
- **Hierarchical Softmax**: Construct a binary tree such that there are $|V|$ leaf nodes each corresponding to one word in the vocabulary.
- There exists a unique path from the root node to a leaf node.

- Let $l(w_1), l(w_2), ..., l(w_p)$ be the nodes on the path from root to w
- Let $\pi(w)$ be a binary vector such that

$$\pi(w)_k = \begin{cases} 1, & \text{path branches left at node } l(w_k) \\ 0, & \text{otherwise} \end{cases}$$

- Finally each internal node is associated with a vector u_i
- So the parameters of the module are $W_{context}$ and $u_1, u_2, ..., u_v$
- The probability of predicting a word is the same as predicting the correct unique path from the root node to that word.
- We model

$$P(\pi(on)_i = 1) = \frac{1}{1 + e^{-v_c^T u_i}} \text{ and } P(\pi(on)_i = 0) = 1 - P(\pi(on)_i = 1)$$

- Note that $p(w|v_c)$ can now be computed using $|\pi(w)|$ computations instead of $|V|$ required by softmax
- Turns out that even a random arrangement of the words on leaf nodes does well in practice.

10.4 GloVe Representations

- Count based methods (SVD) rely on global co-occurrence counts from the corpus for computing word representations
- Predict based methods learn word representations using co-occurrence information
- We will now combine the two.
- Consider X as the SVD decomposed matrix of the previous corpus.
- X_{ij} encodes important global information about the co-occurrence between i and j.
- Now we enforce

$$v_i^T v_j = \log P(j|i) = \log X_{ij} - \log X_i$$
$$v_j^T v_i = \log X_{ji} - \log X_j = \log X_{ij} - \log X_j (X_{ij} = X_{ji})$$

- Adding the two equations we get

$$2v_i^T v_j = 2 \log X_{ij} - \log X_i - \log X_j$$
$$v_i^T v_j = \log X_{ij} - \frac{1}{2} \log X_i - \frac{1}{2} \log X_j = \log X_{ij} - b_i - b_j$$
$$v_i^T v_j + b_i + b_j = \log X_{ij}$$
$$\min_{v_i, v_j, b_i, b_j} \sum_{i,j} (v_i^T v_j + b_i + b_j - \log X_{ij})^2$$

  where $v_i^T v_j + b_i + b_j$ is the predicted value using model parameters and $\log X_{ij}$ is the actual value calculated from the given corpus.

- One drawback is that, this weighs all co-occurrences equally.
- A simple fix is to multiply a weighing function $f(X_{ij})$

$$f(x) = \begin{cases} (\frac{x}{x_{max}})^\alpha, & \text{if } x < x_{max} \\ 1, & \text{otherwise} \end{cases}$$

  where α can be tuned for a given dataset.

10.5 Evaluating Word Representations

- **Semantic Relatedness**
  1. Ask humans to judge the relatedness between a pair of words
  2. Compute the cosine similarity between the corresponding word vectors learned by the model
  3. Given many such word pairs, compute the correlation between S_{model} & S_{human}, and compare different models
  4. Model 1 is better than Model 2 if $correlation(S_{model1}, S_{human}) > correlation(S_{model2}, S_{human})$
- **Synonym Detection**
  1. Given: a term and four candidate synonyms
  2. Pick the candidate which has the largest cosine similarity with the term
  3. Compute the accuracy of different models and compare

11 Sequential Learning

11.1 Introduction

- In many applications the input is not of a fixed size
- Further successive inputs may not be independent of each other
- We need to look at a sequence of (dependent) inputs and produce an output (or outputs)
- Each input corresponds to one time step
- Consider the task of predicting the part of speech tag (noun, adverb, adjective verb) of each word in a sentence
- Once we see an adjective (social) we are almost sure that the next word should be a noun (man)
- Thus the current output (noun) depends on the current input as well as the previous input
- Further the size of the input is not fixed (sentences could have a arbitrary number of words)
- Notice that here we are interested in producing an output at each time step
- Each network is performing the same task (input : word, output : tag)
- Sometimes we may not be interested in producing an output at every stage
- Instead we would look at the full sequence and then produce an output
- consider the task of predicting the polarity of a movie review
- The prediction clearly does not depend only on the last word but also on some words which appear before
- Here again we could think that the network is performing the same task at each step (input : word, output : $+/-$) but it's just that we don't care about intermediate outputs
- Sequences could be composed of anything (not just words)

11.2 Recurrent Neural Networks

- The function being executed at each time step is

$$s_i = \sigma(Ux_i + b)$$
$$y_i = O(Vs_i + c)$$

  where i is the timestep
- Since we want the same function to be executed at each timestep we should share the same network
- This parameter sharing also ensures that the network becomes agnostic to the size of the input
- Since we are simply going to compute the same function at each timestep, the number of timesteps doesn't matter

- We just create multiple copies of the network and execute them at each timestep
- Consider that we feed all previous inputs to the current network, this will be infeasible as now the network does not compute the same function at each time step and hence parameter sharing cannot take place.
- The solution is to add a recurrent connection in the network

$$s_i = \sigma(Ux_i + Ws_{i-1} + b)$$
$$y_i = O(Vs_i + c)$$

- This can be represented as follows

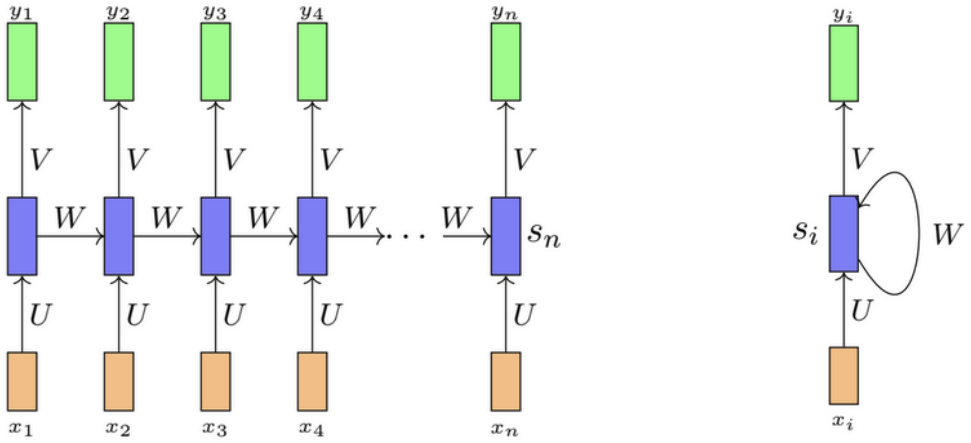


Figure 14: Recurrent Neural Network

- Consider $x_i \in \mathbb{R}^n$, $s_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}^k$, then the dimensions of the parameters are

$$U \in \mathbb{R}^{n \times d}\ V \in \mathbb{R}^{d \times k} \text{ and } W \in \mathbb{R}^{d \times d}$$

- The total loss is simply the sum of loss over all time steps.
- The gradient of V and U is straightforward backpropagation.
- However, gradient of W will be a chain of gradients going through time, aka backpropagation through time.
- For W there is a high chance of vanishing/exploding gradients, as such we restrict backprop time steps to τ.
- The state (s_i) of an RNN records information from all previous time steps
- At each new timestep the old information gets morphed by the current input
- One could imagine that after t steps the information stored at time step $t-k$ gets completely morphed so much that it would be impossible to extract the original information stored at time step $t-k$.
- A similar problem occurs when the information flows backwards
- It is very hard to assign the responsibility of the error caused at time step t to the events that occurred at time step $t-k$

11.3 Long Short Term Memory

- Consider the task of predicting the sentiment of a review
- RNN reads the document from left to right and after every word updates the state
- By the time we reach the end of the document the information obtained from the first few words is completely lost
- Ideally we want to
  1. forget the information added by stop words
  2. selectively read the information added by previous sentiment bearing words
  3. selectively write new information from the current word to the state

- we have computed a state s_{t-1} at timestep $t-1$, and now we want to overload it with new information (x_t) and compute a new state (s_t).
- We introduce a vector o_{t-1} which decides what fraction of each element of s_{t-1} should be passed to the next state
- Each element of o_{t-1} gets multiplied with the corresponding element of s_{t-1}
- Each element of o_{t-1} is restricted to be between 0 and 1

$$o_{t-1} = \sigma(W_o h_{t-2} + U_o x_{t-1} + b_o)$$
$$h_{t-1} = o_{t-1} \odot \sigma(s_{t-1})$$

  W_o, U_o, b_o need to be learned.
- o_t is called the output gate as it decides how much to pass (write) to the next time step
- Now we calculate state information as

$$\tilde{s}_t = \sigma(W h_{t-1} + U x_t + b)$$

- However, we may not want to use all this new information and only selectively read from it before constructing the new cell state s_t
- To do this we introduce another gate called the input gate

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$

  and then take dot product to get new temporary state $\tilde{s}_t$
- To selectively forget, we introduce another forget gate

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

- Now we have the full set of equations for LSTM, the gates are

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$
$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

  and the state equations are

$$\tilde{s}_t = \sigma(W h_{t-1} + U x_t + b)$$
$$s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t$$
$$h_t = o_t \odot \sigma(s_t) \text{ and } rnn_{output} = h_t$$

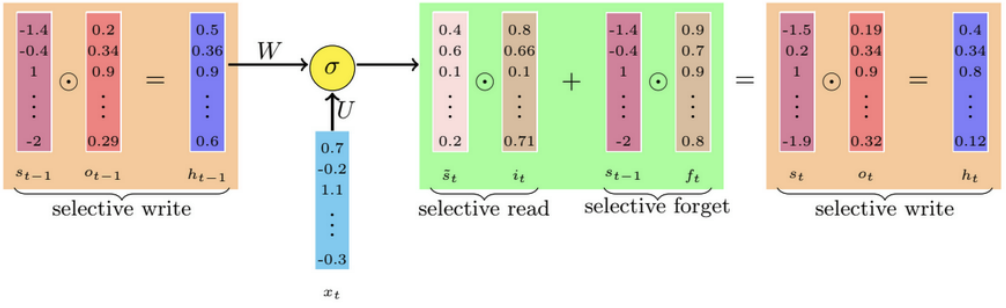


Figure 15: Long Short Term Memory

- Another variant of LSTM is Gated Recurrent Unit, which has the following set of equations

$$o_t = \sigma(W_o s_{t-1} + U_o x_t + b_o)$$
$$i_t = \sigma(W_i s_{t-1} + U_i x_t + b_i)$$
$$\tilde{s}_t = \sigma(W(o_t \odot s_{t-1}) + U x_t + b)$$
$$s_t = (1 - i_t) \odot s_{t-1} + i_t \odot \tilde{s}_t$$

- No explicit forget gate, the forget gate and input gates are tied.
- During forward propagation, the gates control the flow of information
- They prevent any irrelevant information from being written to the state
- Similarly, during backward propagation they control the flow of gradients
- It is easy to see that during backward pass the gradients will get multiplied by the gate
- If the state at time $t-1$ did not contribute much to the state at time t then during backpropagation the gradients flowing into s_{t-1} will vanish
- But this kind of vanishing gradient is fine, since s_{t-1} did not contribute to s_t we don't want to hold it responsible for the crimes of s_t

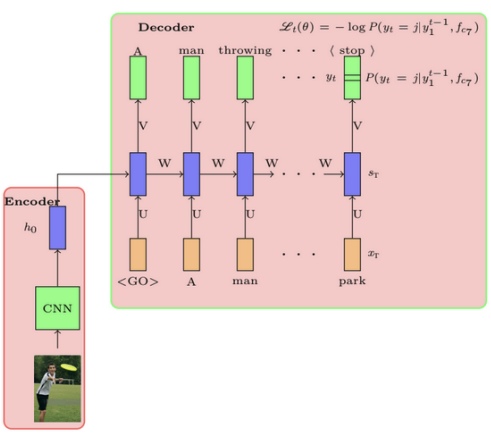
12 Encoder-Decoder Models

12.1 Introduction

- What if we want to generate a sentence given an image?
- We are now interested in $P(y_t|y_{t-1}, I)$ instead of $P(y_t|y_{t-1})$, where I is an image.
- We could now model $P(y_t = j|y_{t-1}, I)$ as $P(y_t = j|s_t, f_{c_7}(I))$ where $f_{c_7}(I)$ is the representation obtained from the convolution layer of an image
- There are many ways of making $P(y_t = j)$ conditional on $f_{c_7}(I)$
- **Option 1**: Set $s_0 = f_{c_7}(I)$
- Now s_0 and hence all subsequent s_t's depend on $f_{c_7}(I)$
- We can thus say that $P(y_t = j)$ depends on $f_{c_7}(I)$
- **Option 2**: Another more explicit way of doing this is to compute

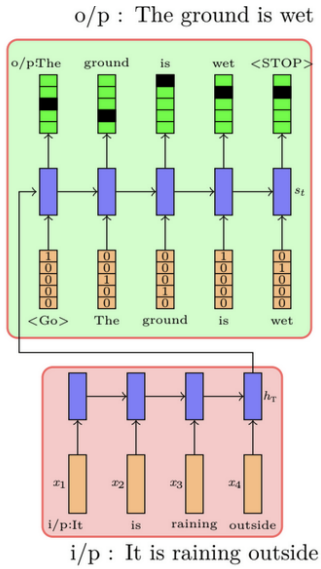
$$s_t = RNN(s_{t-1}, [x_t, f_{c_7}(I)])$$

- We are explicitly using $f_{c_7}(I)$ to compute s_t and hence $P(y_t = j)$
- A CNN is first used to encode the image
- A RNN is then used to decode (generate) a sentence from the encoding
- This is a typical encoder decoder architecture
- Both the encoder and decoder use a neural network
- Alternatively, the encoder's output can be fed to every step of the decoder



- **Task:** Image captioning
- **Data:** $\{x_i = image_i,\ y_i = caption_i\}_{i=1}^{N}$
- **Model:**
  - **Encoder:**
    $$s_0 = CNN(x_i)$$
  - **Decoder:**
    $$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
    $$P(y_t|y_1^{t-1}, I) = softmax(Vs_t + b)$$
- **Parameters:** $U_{dec},\ V,\ W_{dec},\ W_{conv}, b$
- **Loss:**
  $$\mathscr{L}(\theta) = \sum_{i=1}^{T} \mathscr{L}_t(\theta) \quad = -\sum_{t=1}^{T} \log P(y_t = \ell_t|y_1^{t-1}, I)$$
- **Algorithm:** Gradient descent with backpropagation

Figure 16: Image Captioning



- **Task:** Textual entailment
- **Data:** $\{x_i = premise_i,\ y_i = hypothesis_i\}_{i=1}^{N}$
- **Model (Option 1):**
  - **Encoder:**

$$h_t = RNN(h_{t-1}, x_{it})$$

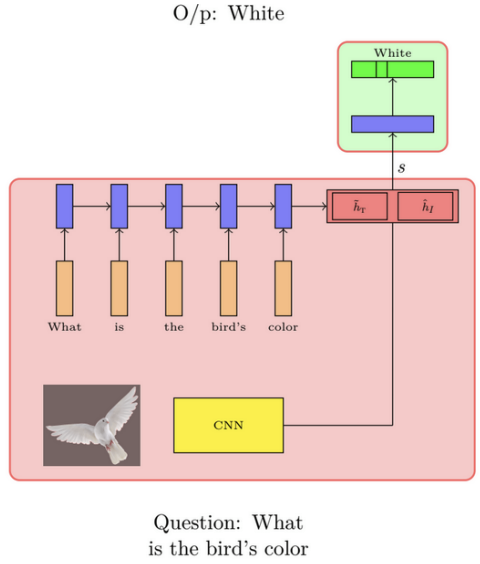
  - **Decoder:**

$$s_0 = h_T \quad (T \text{ is length of input})$$
$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
$$P(y_t|y_1^{t-1}, x) = softmax(Vs_t + b)$$

- **Parameters:** $U_{dec},\ V,\ W_{dec},\ U_{enc},\ W_{enc}, b$
- **Loss:**

$$\mathscr{L}(\theta) = \sum_{i=1}^{T} \mathscr{L}_t(\theta) = -\sum_{t=1}^{T} \log P(y_t = \ell_t|y_1^{t-1}, x)$$

Figure 17: Textual Entailment



- **Task:** Image Question Answeing
- **Data:** $\{x_i = \{I, q\}_i,\ y_i = Answer_i\}_{i=1}^{N}$
- **Model:**
  - **Encoder:**

$$\hat{h}_I = CNN(I),\ \tilde{h}_t = RNN(\tilde{h}_{t-1}, q_{it})$$
$$s = [\tilde{h}_T; \hat{h}_I]$$

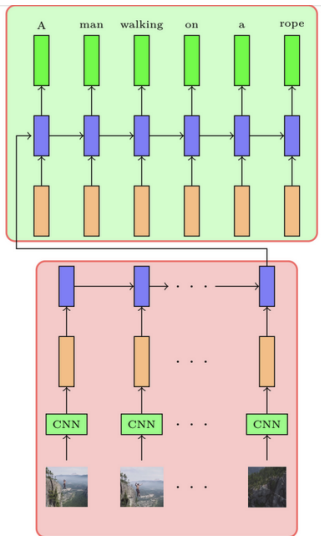
  - **Decoder:**

$$P(y|q, I) = softmax(Vs + b)$$

- **Parameters:** $V,\ b,\ U_q,\ W_q,\ W_{conv}, b$
- **Loss:**

$$\mathscr{L}(\theta) = -\log P(y = \ell|I, q)$$

Figure 18: Image Question Answering



- **Task:** Video Captioning
- **Data:** $\{x_i = video_i,\ y_i = desc_i\}_{i=1}^{N}$
- **Model:**
  - **Encoder:**

$$h_t = RNN(h_{t-1}, CNN(x_{it}))$$

  - **Decoder:**

$$s_0 = h_T$$
$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
$$P(y_t|y_1^{t-1}, x) = softmax(Vs_t + b)$$

- **Parameters:** $U_{dec},\ W_{dec},\ V,\ b,\ W_{conv},\ U_{enc},\ W_{enc}, b$
- **Loss:**

$$\mathscr{L}(\theta) = \sum_{i=1}^{T} \mathscr{L}_t(\theta) = -\sum_{t=1}^{T} \log P(y_t = \ell_t|y_1^{t-1}, x)$$

- **Algorithm:** Gradient descent with backpropagation

Figure 19: Video Captioning

12.2 Attention Mechanism

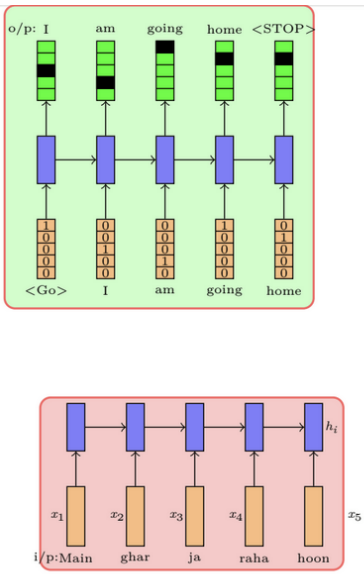
- Encoder decoder models can be made even more expressive by adding an "attention" mechanism
- At each time-step we should feed only this relevant information (i.e. encodings of relevant words) to the decoder
- We could just take a weighted average of the corresponding word representations and feed it to the decoder.
- The machine will have to learn this from the data
- To enable this, we define a function $e_{jt} = f_{ATT}(s_{t-1}, h_j)$
- This quantity captures the importance of the j^{th} input word for decoding the t^{th} output word.
- We can normalize these weights by using the softmax function

$$\alpha_{jt} = \frac{\exp e^{jt}}{\sum_{j=1}^{M} \exp e^{jt}}$$

- α_{jt} denotes the probability of focusing on the j^{th} word to produce the t^{th} output word.
- One possible choice for f_{ATT} is

$$e_{jt} = V_{att}^T \tanh U_{att} s_{t-1} + W_{att} h_j$$

- We are essentially asking the model to approach the problem in a better, more natural way



- **Task:** Machine Translation
- **Data:** $\{x_i = source_i,\ y_i = target_i\}_{i=1}^{N}$
- **Encoder:**

$$h_t = RNN(h_{t-1}, x_t)$$
$$s_0 = h_T$$

- **Decoder:**

$$e_{jt} = V_{attn}^T tanh(U_{attn} h_j + W_{attn} s_t)$$
$$\alpha_{jt} = softmax(e_{jt})$$
$$c_t = \sum_{j=1}^{T} \alpha_{jt} h_j$$
$$s_t = RNN(s_{t-1}, [e(\hat{y}_{t-1}), c_t])$$
$$\ell_t = softmax(Vs_t + b)$$

- **Parameters:** U_{dec}, V, W_{dec}, U_{enc}, W_{enc}, b, U_{attn}, V_{attn}
- **Loss** and **Algorithm** remains same

Figure 20: Architecture with Attention

- We can check whether the attention model learns something meaningful by plotting the attention weights as a heatmap.
- In the case of text, we have a representation for every location of the input sequence.
- But for images, we typically use representation from one of the fully connected layers. This representation does not contain any location information.
- Instead of the f_{c_7} representation, we use the output of one of the convolution layers, which has spatial information.
- For example, the output of the 5^{th} convolutional layer of VGGNet is a $14 \times 14 \times 512$ size feature map. We could think of this as 196 locations, each having a 512 dimensional representation. The model will then learn an attention over these locations.

12.3 Hierarchical Attention

- Consider a dialog between a user (u) and a bot (B)
- The dialog contains a sequence of utterances between the user and the bot
- Each utterance in turn is a sequence of words
- Thus what we have here is a "sequence of sequences" as input.
- We could think of a two level hierarchical RNN encoder
- The first level RNN operates on the sequence of words in each utterance and gives us a representation
- We now have a sequence of utterance representations (red vectors in the image)
- We can now have another RNN which encodes this sequence and gives a single representations for the sequences of utterances
- The decoder can then produce an output sequence conditioned on this utterance

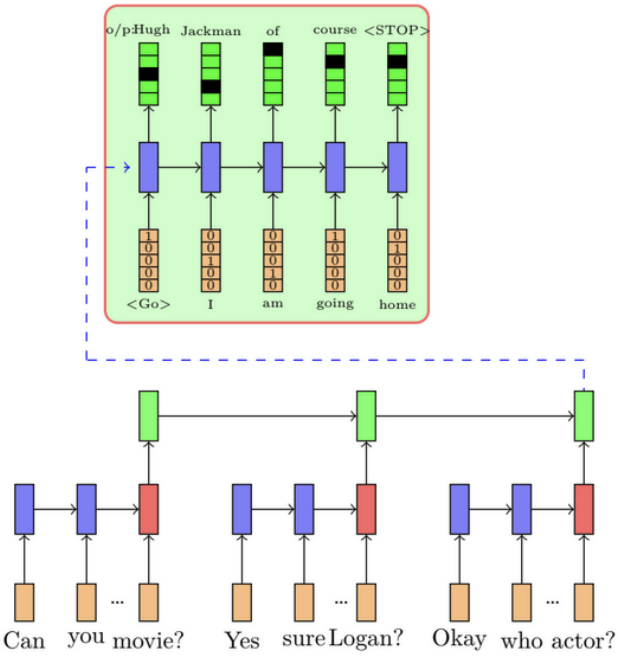
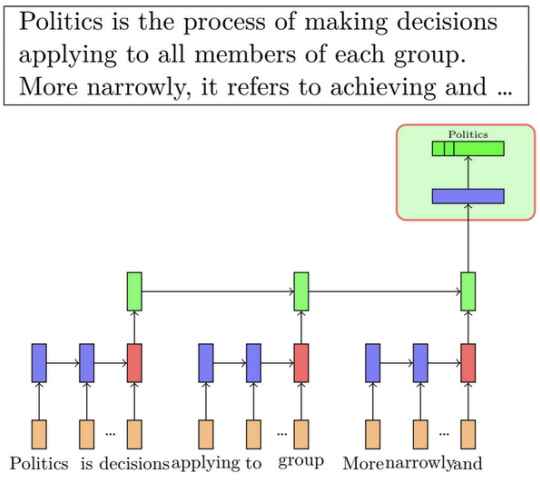


Figure 21: Two layer RNN

- Consider another example of classification of documents, each document is a sequence of sentences and each sentence is in turn a sequence of words.



- **Data:** $\{Document_i, class_i\}_{i=1}^{N}$
- **Word level (1) encoder:**
  $h_{ij}^1 = RNN(h_{ij-1}^1, w_{ij})$
  $s_i = h_{iT_i}^1$ *[T is length of sentence i]*
- **Sentence level (2) encoder:**
  $h_i^2 = RNN(h_{i-1}^2, s_i)$
  $s = h_K^2$ *[K is number of sentences]*
- **Decoder:**
  $P(y|document) = softmax(Vs + b)$
- **Params:** W_{enc}^1, U_{enc}^1, W_{enc}^2, U_{enc}^2, V, b
- **Loss:** Cross Entropy
- **Algorithm:** Gradient Descent with backpropagation

Figure 22: Document Classification

- We need attention at two levels
  First, we need to attend to important (most informative) words in a sentence.
  Then we need to attend to important (most informative) sentences in a document.

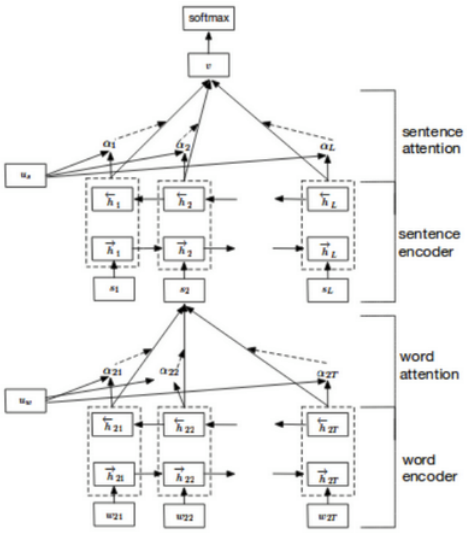


Figure: Hierarchical Attention Network [Yang et al.]

- **Data:** $\{Document_i, class_i\}_{i=1}^{N}$
- **Word level (1) encoder:**

$$h_{ij} = RNN(h_{ij-1}, w_{ij})$$
$$u_{ij} = tanh(W_w h_{ij} + b_w)$$
$$\alpha_{ij} = \frac{exp(u_{ij}^T u_w)}{\sum_t exp(u_{it}^T u_w)}$$
$$s_i = \sum_j \alpha_{ij} h_{ij}$$

- **Sentence level (2) encoder:**

$$h_i = RNN(h_{i-1}, s_i)$$
$$u_i = tanh(W_s h_i + b_s)$$
$$\alpha_i = \frac{exp(u_i^T u_s)}{\sum_i exp(u_i^T u_s)}$$
$$s = \sum_i \alpha_i h_i$$

Figure 23: Encoder Architecture

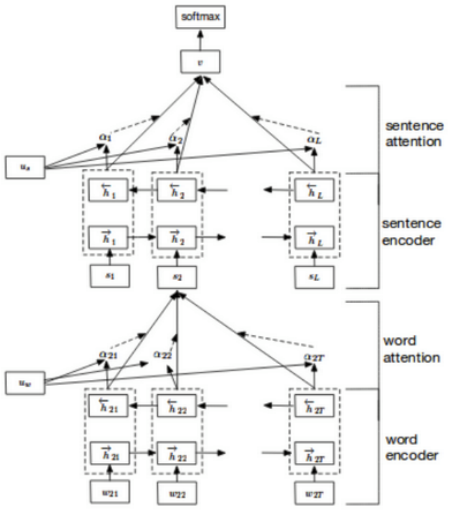


Figure: Hierarchical Attention Network [Yang et al..]

- **Decoder:**

$$P(y|document) = softmax(Vs + b)$$

- **Parameters:**
  W_w, W_s, V, b_w, b_s, b, u_w, u_s
- **Loss:** cross entropy
- **Algorithm:** Gradient Descent and backpropagation

Figure 24: Decoder Architecture

13 Transformer Models

- In encoder-decoder architecture, we have to wait for until s_{t-1} is calculated to calculate s_t.
- The idea is to calculate all these in parallel, instead of waiting.

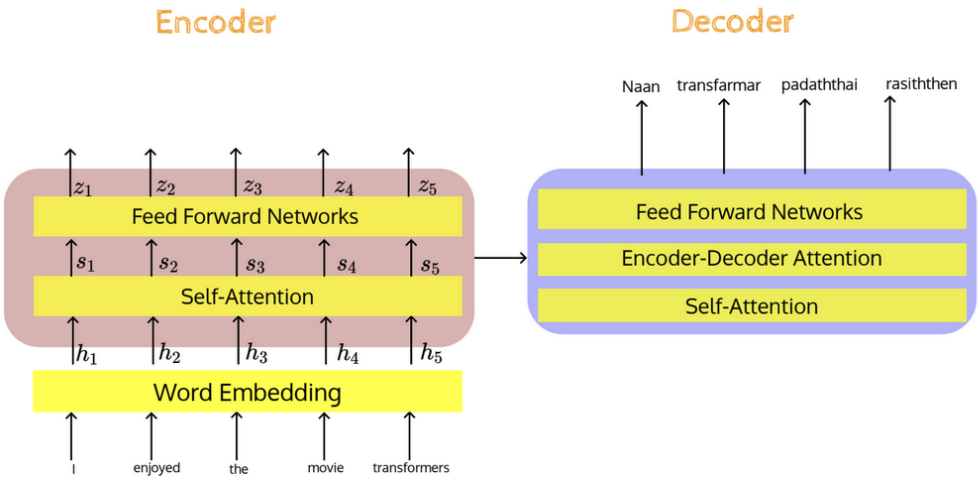


Figure 25: Transformer Architecture

13.1 Self-Attention

- Inputs are vectors, word embeddings, and the outputs are also vectors.
- Given a word in a sentence, we want to compute the relational score between the word and the rest of the words in the sentence, such that the score is higher if they are related contextually.
- now both the vectors s_i and h_j, for all i, j are available for all the time.
- The score function we used was

$$score(s_{t-1}, h_j) = V_{att}^T \tanh U_{att} s_{t-1} W_{att} h_j$$

- There are three vectors involved in computing the score at each time step.
- We get them by transforming h_j
- $q_j = W_Q h_j$, called the query vector
- $k_j = W_K h_j$ called the key vector
- $v_j = W_V h_j$ called the value vector

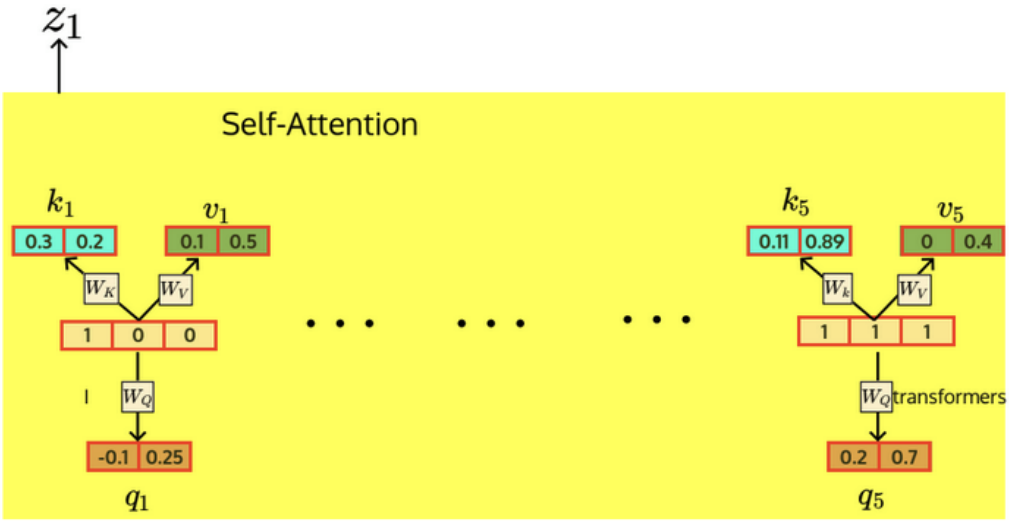


Figure 26: Self Attention

- We will first calculate $score(q_1, k_j)$, $e_{1j} = [q_1 \cdot k_1, q_1 \cdot k_2, ..., q_1 \cdot k_5]$
- Then we get $\alpha_{1j} = softmax(e_{1j})$, finally we get $z_1 = \sum_{j=1}^{5} \alpha_{1j} v_j$
- Query and key vector participate in calculating α, and value vector will participate in getting final output.
- All query, key and value vectors can be computed in one go, as follows

$$Q = [q_1, q_2, ..., q_T] = W_Q[h_1, h_2, ..., h_T]$$

- The entire output can also be computed in parallel as follows

$$Z = [z_1, z_2, ..., z_T] = softmax(\frac{Q^T K}{\sqrt{d_k}})V^T$$

where d_k is the dimension of key vector.

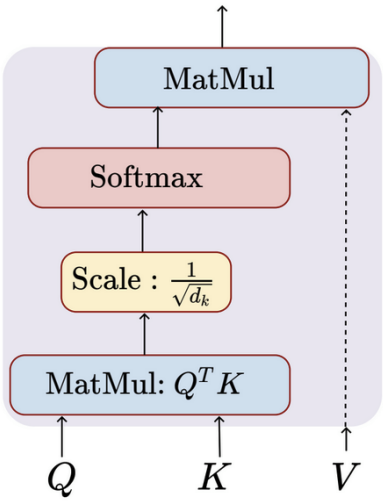


Figure 27: Self Attention Block

13.2 Multi-Headed Attention

- The idea of having multi-headed attention is pretty much the same idea for having multiple filters in CNN.
- **Two-headed attention**: Have two self attention block side-by-side.
- We simply concatenate the output from each block.

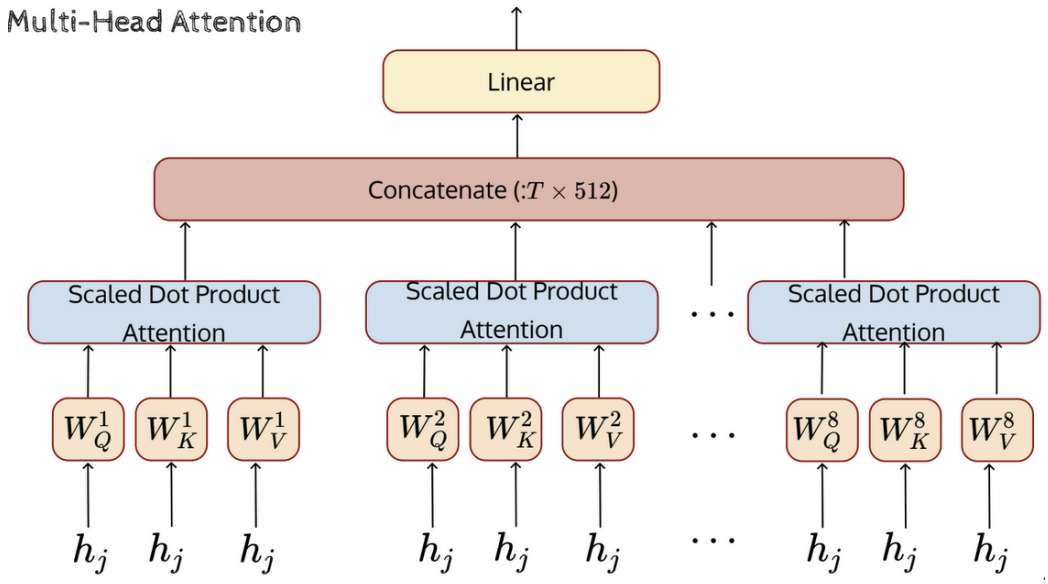


Figure 28: Multi-Headed Attention Block

- The outputs are then passed to a feed forward network, this is our usual ANN.

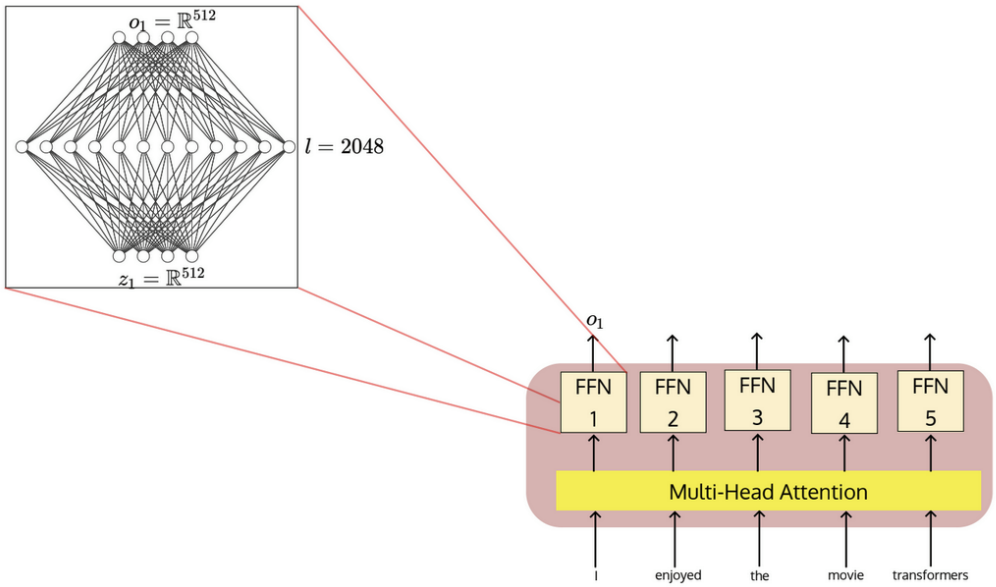


Figure 29: Feed Forward Network

- Identical network for each position, $FFN(z) = \max(0, W_1 z + b_1)W_2 + b_2$
- The encoder is composed of identical layers, and each layer is composed of 2 sub-layers.
- The computation is parallelized in the horizontal direction, i.e. within a training sample, of the encoder stack, not along the vertical direction.

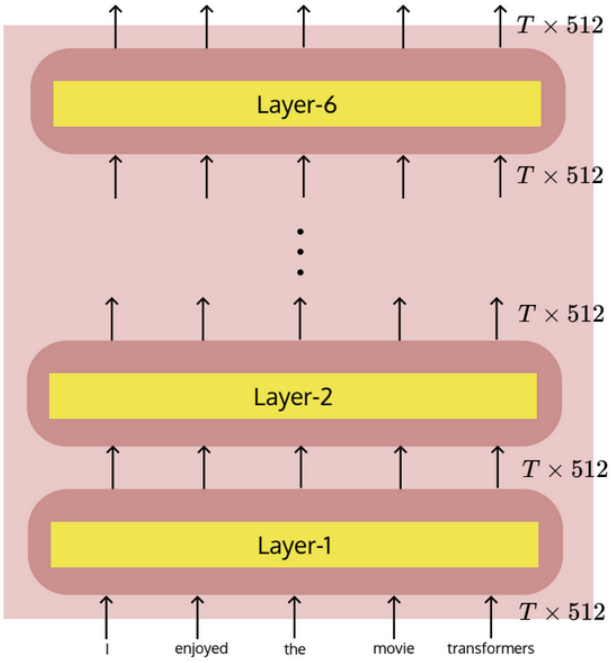


Figure 30: Encoder Stack

13.3 Decoder Architecture

- Takes input from the encoder, and also has self-inputs, they are the words that we have predicted/decoded so far.
- Each layer is composed of three sub-layers.

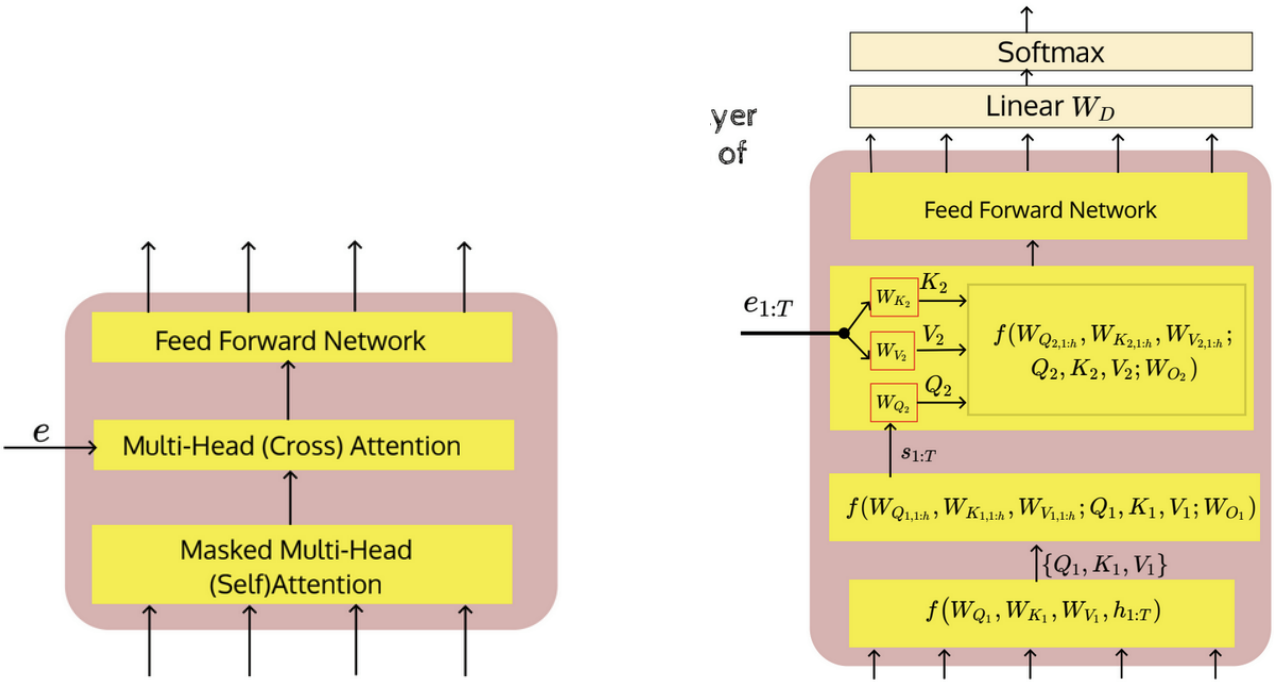


Figure 31: Decoder Block

- We don't know how long is the output going to be, hence we will consider a max length T_1.
- At the beginning only $< GO >$ makes sense, all other inputs will be junk, hence we have "masked" self attention. Essentially, we make all α greater than current decoded word 0.
- Masking is done by inserting negative infinite at the respective positions. This works because we take max of values(ReLU) in the feedforward network.
- We will also use masking in cross attention.
- We will take the values coming from encoder, and get K and V. We will use output from self-attention as Q, then cross attention becomes the same as self-attention.