সূচিপত্র

- পরিচিতি 1.1
- মেশিন লার্নিং বনাম ডিপ লার্নিং 1.2
- আর্টিফিশিয়াল নিউরাল নেটওয়ার্ক 1.3
- সহজ একটি নিউরাল নেটওয়ার্ক 1.4
  - ট্রেনিং এর পদ্ধতি 1.4.1
  - ফিরে দেখা 1.4.2
  - কোডিং 1.4.3
  - পরীক্ষা করে দেখা 1.4.4
- মাল্টি লেয়ার নিউরাল নেটওয়ার্ক 1.5
  - কোডিং 1.5.1
  - পরীক্ষা করে দেখা 1.5.2
- কনভলিউশনাল নিউরাল নেটওয়ার্ক 1.6
  - ব্যাসিক পরিচিতি 1.6.1
  - কনভলিউশন করা 1.6.2
  - ফুলি কানেক্টেড লেয়ার 1.6.3
  - কমপ্লিট কনভলিউশনাল নিউরাল নেটওয়ার্ক 1.6.4
  - পরীক্ষা করে দেখা 1.6.5
- বিভিন্ন লাইব্রেরীর ব্যবহার 1.7
- TensorFlow পরিচিতি 1.8
  - TensorFlow ব্যাসিক 1.8.1
  - ভ্যারিয়েবল ও প্লেসহোল্ডার 1.8.2
- TensorFlow দিয়ে ইমেজ ক্লাসিফায়ার তৈরি 1.9
  - ডাটা বুঝে নেয়া 1.9.1
  - ডাটা ডাইমেনশন 1.9.2
  - গ্রাফ তৈরি 1.9.3
  - মডেল 1.9.4
  - Cost ফাংশন ও অপটিমাইজেশন 1.9.5
  - TensorFlow রান 1.9.6
- TensorFlow দিয়ে কনভলিউশনাল NN 1.10
- Pretty Tensor 1.11

Inception মডেল 1.12
ট্রান্সফার লার্নিং 1.13
ডিপ ড্রিম 1.14
রি-ইনফোর্সমেন্ট লার্নিং 1.15

ডিপ লার্নিং ও আর্টিফিশিয়াল নিউরাল নেটওয়ার্ক

কোর্স পরিচালনায় **(Lead Author)**
Nuhil Mehdy

স্বয়ংক্রিয় কন্ট্রিবিউটরের তালিকা
(প্রথম ৫ জন)

[013] Nuhil Mehdy
[001] gitbook-bot

ভূমিকা
দেরি করে হলেও ডিপ লার্নিং এর ব্যবহার ও প্রয়োজনীয়তা ইদানীং ব্যাপক হারে বাড়ছে। কম্পিউটার ভিশন, ন্যাচারাল ল্যাঙ্গুয়েজ প্রসেসিং সহ বেশ কিছু সেক্টরে এর প্রভাব লক্ষণীয়। ডিপ লার্নিং হচ্ছে মেশিন লার্নিং এর একটি ব্র্যাঞ্চ বা একটা মেশিন লার্নিং টেকনিক যা কিনা নিজে নিজেই সরাসরি ডাটা থেকে ফিচার এবং টাস্ক শিখে নিতে পারে। সেই ডাটা হতে পারে ইমেজ, টেক্সট এমনকি সাউন্ড। অনেকেই ডিপ লার্নিং -কে এন্ড টু এন্ড লার্নিং-ও বলে থাকেন। ডিপ লার্নিং টেকনিকের খুব পুরনো এবং বহুল পরিচিত ব্যাবহার হয় পোস্টাল সার্ভিসে খামের উপর বিভিন্ন ধরনের হাতের লেখা চিহ্নিত করতে। মোটামুটি ১৯৯০ সালের দিক থেকেই ডিপ লার্নিং এর এই প্রয়োগ চলে আসছে।

২০০৪/২০০৫ সালের দিক থেকে ডিপ লার্নিং এর ব্যবহার খুব উল্লেখ যোগ্য ভাবে বেড়ে চলছে। মূলত তিনটি কারণে —প্রথমত ইদানিং কালের ডিপ লার্নিং মেথড গুলো মানুষের চেয়ে অনেক বেশি ভালো ভাবে অবজেক্ট রিকগনিশনের বা ক্লাসিফিকেশনের কাজ করতে পারছে, দ্বিতীয়ত GPU এর কল্যাণে অনেক বড় আকারের ডিপ নেটওয়ার্ক খুব কম সময়ের মধ্যেই লার্নিং শেষ করে নিতে পারছে, তৃতীয়ত, খুব ইফেক্টিভ লার্নিং এর জন্য যে পরিমাণ ডাটার প্রয়োজন পরে সেই লেভেলের ডাটা গত ৫/৬ বছরে ব্যবহার উপযোগীভাবে তৈরি হচ্ছে বিভিন্ন সার্ভিসের মাধ্যমে।

বেশির ভাগ ডিপ লার্নিং মেথড নিউরাল নেটওয়ার্ক আর্কিটেকচার ফলো করে আর তাই ডিপ লার্নিং মডেলকে মাঝে মধ্যেই ডিপ নিউরাল নেটওয়ার্ক হিসেবেও বলা হয়ে থাকে। খুব পপুলার একটি ডিপ লার্নিং মডেল হচ্ছে কনভলিউশনাল নিউরাল নেটওয়ার্ক বা CNN. এ ধরনের নেটওয়ার্ক বিশেষ করে ইমেজ ডাটা নিয়ে কাজ করার সময় ব্যবহৃত হয়ে থাকে। যখন বেশ কিছু লেয়ার নিয়ে একটি নিউরাল নেটওয়ার্ক ডিজাইন করা হয় তখন এটাকেই ডীপ নিউরাল নেটওয়ার্ক বলে। এই লেয়ারের সংখ্যা হতে পারে ২-৩ টি থেকে শ-খানেক পর্যন্ত।

এ পর্যন্ত পড়ার পর যদি খুব অস্বস্তি চলে আসে তবে ভয় পাওয়ার কিছু নাই, নিচেই খুব ব্যাসিক কিছু উদাহরণ এর মাধ্যমে সব সহজ ভাবে আলোচনা করা হবে। আমরা একটা সমস্যা দেখবো এবং তার সমাধানের জন্য একটি নিউরাল নেটওয়ার্ক ডিজাইন করবো। তারপর পাইথনে কোড লিখে সেই নেটওয়ার্কের প্রোগ্রামেটিক ভার্শন লিখবো এবং সেটার লার্নিং করিয়ে সমস্যাটা সমাধানও করবো ইনসা আল্লাহ। তার আগে পরবর্তী চ্যাপ্টারে জেনে নেব, মেশিন লার্নিং এবং ডিপ লার্নিং এর মধ্যে পার্থক্য বা সম্পর্ক কোথায়।

ওপেন সোর্স

এই বইটি মূলত স্বেচ্ছাশ্রমে লেখা এবং বইটি সম্পূর্ন ওপেন সোর্স । এখানে তাই আপনিও অবদান রাখতে পারেন লেখক হিসেবে । আপনার কন্ট্রিবিউশান গৃহীত হলে অবদানকারীদের তালিকায় আপনার নাম স্বয়ংক্রিয়ভাবে যুক্ত হয়ে যাবে।

এটি মূলত একটি গিটহাব রিপোজিটোরি যেখানে এই বইয়ের আর্টিকেল গুলো মার্কডাউন ফরম্যাটে লেখা হচ্ছে । রিপোজিটরিটি ফর্ক করে পুল রিকুয়েস্ট পাঠানোর মাধ্যমে আপনারাও অবদান রাখতে পারেন । বিস্তারিত দেখতে পারেন এই ভিডিওতে Video

> বর্তমানে বইটির কন্টেন্ট বিভিন্ন কন্ট্রিবিউটর এবং নানা রকম সোর্স থেকে সংগৃহীত এবং সংকলিত।

Disclaimer: ডিপ লার্নিং এর একদম ব্যাসিক ফাংশনালিটি, এর সাথে সম্পর্কিত বিভিন্ন টার্ম গুলোর পরিচয় এবং ডিপ লার্নিং সম্বন্ধে গুরুগম্ভীর লেখা/বই পড়তে/বুঝতে পারার উপযোগী পাঠক তৈরি করাই এই কোর্সের উদ্দেশ্য। আমি ইন্টারনেটের বিভিন্ন সোর্স থেকে পড়ে, দেখে যে ব্যাসিক ধারনাটা পেয়েছি সেগুলোই গুছিয়ে এক জায়গায় করে অন্যদের সাথে শেয়ার করবো এই কোর্সে। কোর্সের শেষে সব গুলো রেফারেন্স জুড়ে দেয়া হবে।

মেশিন লার্নিং বনাম ডিপ লার্নিং

ইতোমধ্যে একটা ধারনা পেয়ে গেছেন যে, ডিপ লার্নিং এর মাধ্যমে বেশিরভাগ সময়েই ডাটা ক্লাসিফিকেশনের কাজ করা হয়ে থাকে। কিন্তু যদি আপনার মেশিন লার্নিং সম্পর্কে ধারনা থেকে থাকে তাহলে হয়ত এটাও জানেন যে, মেশিন লার্নিং এর বিভিন্ন অ্যালগরিদম ব্যবহার করেও ডাটা ক্লাসিফিকেশনের কাজ করা যায়। কিন্তু দুটো মাধ্যমের কাজ করার ধরনে বেশ কিছু পার্থক্য আছে। যেমন- মেশিন লার্নিং ব্যবহার করে ইমেজ ক্লাসিফিকেশনের কাজ করার সময় প্রথমেই ট্রেনিং ডাটা(ইমেজ) থেকে আপনার নিজেকেই ফিচার (যেমন-এইজ, কর্নার ইত্যাদি) এক্সট্র্যাক্ট করে নিতে হবে আপনার মেশিন লার্নিং মডেলকে ট্রেনিং করানোর জন্য। এরপর নতুন ডাটা(ইমেজ) নিয়ে সেটার ধরন প্রেডিক্ট করার সময় আপনার মেশিন লার্নিং মডেল সেই ফিচার গুলোকেই ভ্যারিয়েবল (চেনার হাতিয়ার) হিসেবে কাজে লাগিয়ে নতুন ইমেজটাকে অ্যানালাইস করে ডিসিশন নেয়। এভাবে ইমেজের মধ্যে কোন একটি নির্দিষ্ট অবজেক্ট রিকগনিশন বা ডিটেকশণ এর কাজও করা হয়ে থাকে।

অন্যদিকে ডিপ লার্নিং টেকনিকে কাজ করার সময় আপনি ম্যানুয়ালি সেই ফিচার এক্সট্র্যাকশনের কাজ থেকে বেঁচে যেতে পারেন। এক্ষেত্রে আপনি পুরো ইমেজটাকেই আপনার ডিপ নিউরাল নেটওয়ার্কে ইনপুট হিসেবে দিয়ে দিতে পারেন এবং সেই নেটওয়ার্ক স্বয়ংক্রিয় ভাবেই লেবেলের সাথে সম্পর্ক রেখে ওই ইমেজের গুরুত্বপূর্ণ ফিচারগুলোকে কন্সিডার করে লার্নিং করে নিতে পারবে যাতে করে সে পরবর্তীতে নতুন ইমেজ থেকে একই ধরনের অবজেক্ট খুঁজে নিতে পারে বা তার টাস্ক সম্পন্ন করতে পারে।

কখন আপনার জন্য মেশিন লার্নিং অ্যাপ্রোচ ভালো হবে এবং কখন আপনি ডিপ লার্নিং নিয়ে কাজ করলে সুবিধা হবে সেটা নির্ভর করে আপনার নির্দিষ্ট সমস্যাটির টাইপের উপর এবং আপনার কাছে থাকা ডাটার পরিমাণ ও ধরনের উপর। তবে খুব সহজ ভাবেও প্রাথমিক একটা সিদ্ধান্ত নিতে পারেন। যেমন- আপনার কাছে যদি তুলনামূলক কম পরিমাণ ডাটা থাকে এবং আপনার কম্পিউটেশন পাওয়ারও সীমাবদ্ধ হয় তাহলে আপনার জন্য মেশিন লার্নিং অ্যাপ্রোচ ভালো হবে। এতে করে আপনি একদিকে কম রিসোর্স ব্যবহার করেই সমস্যাটি নিয়ে কাজ করতে পারবেন এবং সাথে সাথে যেহেতু আপনি নিজেই ডাটা থেকে ফিচার পছন্দ করেন আর বর্তমানে অনেক গুলো মেশিন লার্নিং অ্যালগরিদম আছে তাই, বিভিন্ন ফিচার এবং অ্যালগরিদম এর কম্বিনেশন ব্যবহার করে আপনার মডেল এর পারফর্মেন্স চেক করে দেখতে পারেন।

অন্যদিকে যদি আপনার কাছে অনেক পরিমাণ ডাটা থাকে এবং সাথে সাথে আপনার কাছে যথেষ্ট পরিমাণ কম্পিউটেশন পাওয়ার থাকে তাহলে আপনার জন্য ডিপ লার্নিং অ্যাপ্রোচ ভালো হবে। এতে করে অনেক অনেক ডাটা থেকে ট্রেনিং করানোর সময় আপনার ডিজাইন করা নিউরাল নেটওয়ার্কটি অনেক বেশি পারফেকশন দেখাতে পারবে। এমনকি সেই ডাটা গুলো থেকে ধরে ধরে আপনাকে ফিচার পছন্দ না করে দিলেও চলবে। তবে এর জন্য মারাত্মক রকম কম্পিউটেশন পাওয়ার এবং সময়ও দরকার পরবে।

আর্টিফিশিয়াল নিউরাল নেটওয়ার্ক কি?

নিউরাল নেটওয়ার্ক এবং অন্য যেকোনো রকম নেটওয়ার্ক (যেমন – কিছু কম্পিউটার মিলে একটি লোকাল এরিয়া নেটওয়ার্ক অথবা পুরো ওয়েব নেটওয়ার্ক) বস্তুত একই। বেশ কিছু নোড বা পয়েন্ট একে ওপরের সাথে নির্দিষ্ট কিছু নিয়মে যুক্ত থেকে নিজেদের মধ্যে তথ্য আদান প্রদান করলেই তাকে একটা নেটওয়ার্ক বলা যায়। নিউরাল নেটওয়ার্কের ক্ষেত্রে সেই নোড (Node) হচ্ছে এক একটি নিউরন। আমাদের ব্রেইনের মধ্যে বস্তুত বিলিয়ন সংখ্যক নিউরনের একটা নেটওয়ার্ক তৈরি করা আছে। মোটামুটি সেই গঠন শৈলীর উপর ভিত্তি করেই ডাটা থেকে প্যাটার্ন রিকগনিশনের জন্য এক ধরনের কার্যপদ্ধতির নামই হচ্ছে আর্টিফিশিয়াল নিউরাল নেটওয়ার্ক। অর্থাৎ সত্যিকারের নিউরান যে নীতিতে কাজ করে, এই নিউরনও একইভাবে কাজ করে। কিন্তু যেহেতু এগুলো সত্যিকারের নিউরন নয় তাই এটার নাম আর্টিফিশিয়াল নিউরাল নেটওয়ার্ক।

ভিত্তি

তো আমরা যদি একটু দুঃসাহস করে সত্যিকারের একটা নিউরনের কার্যনীতি দেখি তাহলে আমরা জানতে পারি যে – একটা নিউরনের কিছু ইনপুট দরকার এবং সেই ইনপুট গুলো আসে Dendrite নামের কিছু ডাল পালার মত অংশ দিয়ে, এরপর নিউরন বডি বা Soma নামের অংশে কিছু ক্যালকুলেশন হয় সেই ইনপুট গুলোর উপর। অতঃপর Axon নামের লেজের মত একটা অংশ দিয়ে সেই ক্যালকুলেশনের আউটপুট বের হয় যা কিনা আবার অন্য এক বা একাধিক নিউরনের ইনপুট স্লট তথা Dendrite এ চলে যায়। একটি নিউরনের এক্সন এবং অন্য নিউরনের ডেন্ড্রাইটের মাঝে Synapse নামের কিছু তরল থাকে। এটাই বস্তুত এক নিউরন থেকে আরেক নিউরনের কাছে এক্সনের আউটপুট কে ডেন্ড্রাইটে ইনপুট দেয়ার ক্ষেত্রে ট্রান্সমিশনের ভূমিকা রাখে। যদি একটি নিউরনের যথেষ্ট পরিমাণ সিন্যাপ্টিক ইনপুট ফায়ার (আশানুরূপ একটা ভ্যালু তৈরি করে) করে তাহলে সেই নিউরনটা ফায়ার করে বা বলা যেতে পারে যে, সেই নিউরনটা অ্যাকটিভ হয়। বিষয় হচ্ছে – এই ঘটনাকেই চিন্তা করা বলে।

বানিয়ে ফেলি একটা নিউরাল নেটওয়ার্ক

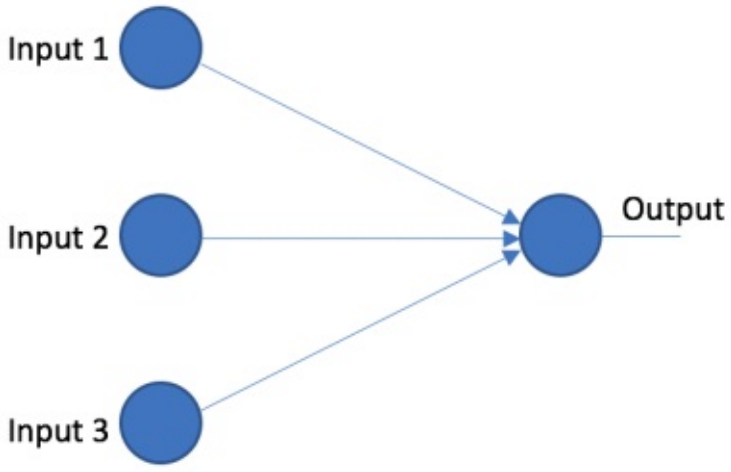
উপরোক্ত প্রসেসটাকেই আমরা কম্পিউটারে প্রোগ্রাম লিখে সিমুলেট করতে পারি। তাহলেই কিন্তু একটা আর্টিফিশিয়াল নিউরন বানাতে পারছি বলে ধরে নয়া যায়। তাই না? তবে এটা সত্যিকারের নিউরনের মত জটিল হবে না কারণ, আগেই বলা হয়েছে যে আর্টিফিশিয়াল নিউরাল নেটওয়ার্ক শুধুমাত্র সত্যিকারের নিউরনের নেটওয়ার্কের কনসেপ্ট এর উপর ভিত্তি করে তৈরি। তার মানে এই না যে এটা শতভাগ একই। যাই হোক আমরা একটি সিঙ্গেল নিউরন তৈরি করি যার মাধ্যমেই আমরা ছোট্ট একটা সমস্যা সমাধানের চেষ্টা করবো। ধরে নেই আমাদের আর্টিফিশিয়াল ব্রেনে একটাই মাত্র নিউরন।

সমস্যাটা নিচের মত,

	ইনপুট কম্বিনেশন			আউটপুট
উদাহরণ ১	0	0	1	0
উদাহরণ ২	1	1	1	1
উদাহরণ ৩	1	0	1	1
উদাহরণ ৪	0	1	1	0
নতুন অচেনা ইনপুট কম্বিনেশন	1	0	0	?

উপরের প্যাটার্ন গুলো নিয়ে একটু চিন্তা করলেই আমরা বুঝতে পারবো যে প্রত্যেকটি ইনপুট কম্বিনেশনের প্রথম ভ্যালুটি বস্তুত ওই কম্বিনেশনের আউটপুট। পরের দুটি ভ্যালু অনর্থক। তাই নতুন অচেনা কম্বিনেশন 1 0 0 এর আউটপুট হবে 1. আমরা নিজেরা এটা সহজেই চিন্তা করে ধরে ফেলেছি। এই কাজটাই আমরা আরটিফিশিয়াল নিউরনের মাধ্যমে করে দেখতে চাই।

এখন আমাদের আর্টিফিশিয়াল ব্রেনকে উপরের চারটা উদাহরণ (কিছু ইনপুট কম্বিনেশন এবং তার সাপেক্ষে একটি করে আউপুটপুট) দেখিয়ে বললাম এটার প্যাটার্ন শিখে নাও যাতে করে একই প্যাটার্নেরই কিন্তু নতুন অচেনা একটা ইনপুট কম্বিনেশনের আউটপুট কত হবে সেটা বলতে পারো। ধরে নেই আমাদের নিউরন দেখতে নিচের মত।



এই নিউরনেরও তিনটা ইনপুট নেয়ার জন্য তিনটা ডেন্ড্রাইট আছে আর একটা এক্সন আছে যেখানে আমরা আউপুট পেয়ে যাবো কারণ এই ব্রেনে আর কোন নিউরন নাই যাকে এই এক্সনের আউপুটকে ইনপুট হিসেবে দিতে হবে।

কিভাবে এই নিউরনকে ট্রেনিং দেয়া যায়?

আমরা যেটা করতে পারি – নিউরনটার তিনটা ইনপুটে আমাদের প্রথম উদাহরণের তিনটা ভ্যালু দেবো আর আউপুট প্রান্তে সেই উদাহরণ মোতাবেক সঠিক আউপুটটা রাখবো। এরপর ইনপুট লাইন (গ্রাফ কনসেপ্টে Edge) গুলোতে কিছু Weight (ভ্যালু) সেট করবো। এরপর প্রত্যেকটা ইনপুট ভ্যালুর সাথে ওই Edge এর ভ্যালু তথা Weight গুন করে নিউরন-বডি/Soma বা Node এ একটা Weighted Sum জমা করবো। এক্ষেত্রে এই Node টী হচ্ছে আমাদের আউপুট নোড। যাই হোক, এই নোডে জমা হওয়া ভ্যালুর পরিমাণ কিন্তু 1 এর অনেক বেশি বা 0 এর চেয়ে কম হতে পারে। কিন্তু আমরা তো চাই, ভ্যালুটা 1 আর 0 এর মাঝা মাঝি থাকুক যাতে করে আমরা আমাদের আগে থেকেই সেট করা আউপুট ভ্যালু (যেমন – প্রথম উদাহরণ মোতাবেক 0) –এর সাথে তুলনা করতে পারি। তাই আউটপুট নোডে জমা হওয়া ভ্যালুকে আমরা গণিতের একটা স্পেশাল গ্যারাকলে ফেলে কোন না কোন ভাবে 1 ও 0 এর মাঝেই রাখবো। এটাকে বলে Activation Function (নিচেই আমরা এর চেহারা এবং ব্যবহার দেখবো)। এরপর সেই ক্যালকূলেটেড ভ্যালু এবং প্রথম উদাহরণ মোতাবেক আউটপুটের ভ্যালুর মধ্যে তুলনা করে আমরা ভুলের পরিমাণ দেখবো।

অর্থাৎ প্রথম ট্রেনিং হচ্ছে – ইনপুট দিলাম 0 0 1 এবং আউটপুট যেন হয় 0. কিন্তু ধরি Edge গুলোতে সেট করা আমাদের র‍্যান্ডম ওয়েট গুলো কম্বিডার করে ওয়েটেড সাম থেকে অ্যাক্টিভেশন ফাংশনের মাধ্যমে ভ্যালু পেলাম 0.9

তার মানে ট্রেনিং এর মাথা মুণ্ড কিছুই হয় নি। এরপর আমরা যা করতে পারি সেটা হল, সেই ওয়েট গুলোকে একটু চেঞ্জ (Adjust) করে দেখতে পারি। অর্থাৎ প্রথম লাইনে (Edge) যে ভ্যালু ছিল সেটাকে একটু বাড়িয়ে, আবার দ্বিতীয় লাইনের ওয়েটকে একটু কমিয়ে আবারও Activation Function এর আউটপুট দেখতে পারি। এবার যদি দেখি যে এর মান আসলো 0.4 তার মানে আউটপুটের সাথে মিল আসতেছে। আবারও একটু ওয়েট গুলোকে অ্যাডজাস্ট করে Activation Function এর আউটপুট দেখতে পারি। এবার যদি দেখি যে এর মান আসলো 0.1 তার মানে প্রথম ট্রেনিং সেট মোতাবেক যে 0 পাওয়ার চেষ্টা সেটা মোটামুটি সফল।

অর্থাৎ প্রথম ডাটা সেট (একটা Row যার ইনপুট/ভ্যারিয়েবল 0 0 1 এবং আউটপুট/লেবেল 0) এর জন্য আমাদের নিউরনের ট্রেনিং সম্পন্ন। অর্থাৎ, নিউরনটি তার ওয়েট গুলোকে অ্যাডজাস্ট করে এমন একটা অবস্থায় নিয়ে এসেছে যে এর তিনটি ইনপুটে 0 0 1 দিলে আউটপুটে 0 এর কাছাকাছি একটা ভ্যালু আসে। এরপর আবার নিউরনকে দ্বিতীয় ট্রেনিং ডাটা সেট দেয়া হল। এবার তার কাজ হচ্ছে নতুন তিনটি ইনপুট 1 1 1 এবং এর জানা আউটপুট 1 এর সাপেক্ষে নিজের Edge গুলোর ওয়েট এমনভাবে অ্যাডজাস্ট করা, যাতে করে ওয়েটেড সাম এবং Activation Function এর কাজের পর আউটপুট মোটামুটি 1 এর কাছাকাছি আসে।

কিন্তু খেয়াল রাখতে হবে যে আগের ট্রেনিং সেটের ব্যাপারটাও যাতে ঠিক থাকে। অর্থাৎ, দ্বিতীয় ট্রেনিং সেটের জন্য ওয়েট অ্যাডজাস্ট করে ইনপুট আউটপুট মিলাতে গিয়ে যেন প্রথম ট্রেনিং সেটের বারোটা না বেজে যায়। তার মানে তাকে খুবি ধীরে সুস্থে এবং ক্ষুদ্র ক্ষুদ্র পরিমাণে ওয়েটের ভ্যালু অ্যাডজাস্ট করতে হবে যাতে 0 0 1 দিলেও 0 এর মত মান আসে আবার 1 1 1 দিলেও যাতে 1 এর মত মান আসে। তো, বোঝাই যাচ্ছে যে, একবার দুইবার অ্যাডজাস্ট করে এই কাজ হাসিল করা সম্ভব না। বার বার এরর চেক করে বার বার ওয়েট অ্যাডজাস্ট করে করে দেখতে হবে। মাথা গরম করার কিছু নাই, নিচে আবারও আমরা এসব কথা বার্তা আরেকবার ধাপে ধাপে দেখবো।

তবে বুঝতে পারছি যে ঠিক কি কি কাজ আমাদের করতে হবে একটা ট্রেনিং সাইকেল সম্পন্ন করতে হলেঃ

এক) ট্রেনিং ডাটা টেবিল থেকে একটা সেট নিয়ে ইনপুট গুলো দেবো। এইজ গুলোতে কিছু র‍্যান্ডম ওয়েট (ভ্যালু) সেট করবো। কিছু গুন আর যোগ করে এবং স্পেশাল একটা ফাংশনের মাধ্যমে এর আউটপুট বের করবো।

দুই) এরর এর পরিমাণ বের করবো অর্থাৎ – এই ধাপের আউটপুট এবং আসলেই ট্রেনিং সেট মোতাবেক আসল আউটপুটের পার্থক্য দেখবো।

তিন) এরর এর গতবিধি মোতাবেক ওয়েটগুলোকে খুব অল্প পরিমাণে অ্যাডজাস্ট করবো

চার) উপরের তিনটি ধাপকে হাজার হাজার বার রিপিট করাবো

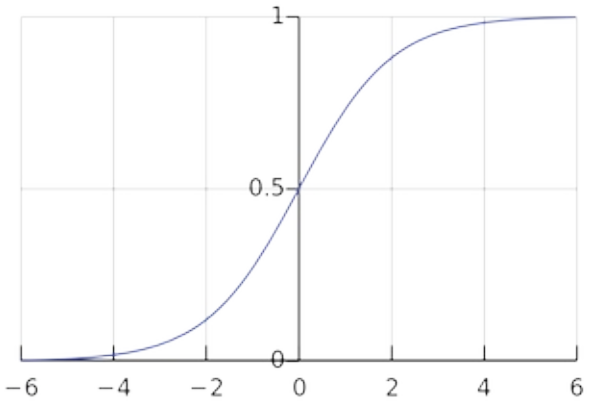
এভাবে সামনের দিকে ক্যালকুলেশন এগিয়ে নিয়ে (Forward Propogation) এরর সাথে তুলনা করে আবার পিছনে ফিরে এসে (Back Propogation) ভ্যালু (ইনপুট ভ্যালু না কিন্তু। ওয়েট বা আপনার সেট করা ভ্যালু) গুলোকে অ্যাডজাস্ট করে আবার আউটপুটের সাথে তুলনা করার যে চক্র তাকে বলে Back Propogation.

এক নাম্বার ধাপের কাজ করি – প্রত্যেকটা ইনপুটের সাথে প্রত্যেক লাইনের ওয়েট গুন করে যোগফল বের করার সূত্র হবে নিচের মতঃ

$$\sum weight_i \cdot input_i = weight1 \cdot input1 + weight2 \cdot input2 + weight3 \cdot input3$$

এর পর এই যোগফলকে 1 ও 0 এর মাঝা মাঝি রাখার জন্য গণিতের একটা স্পেশাল ফাংশন যার নাম Sigmid Function (এখানে এটাই আমাদের Activation Function. এরকম আরও আছে।) সেটাকে ব্যবহার করতে পারি। এই ফাংশনের কাজ হচ্ছে – একে ইনপুট হিসেবে যে মাণই দেয়া হোক না কেন, আউটপুট আসবে 1 থেকে শুনের মধ্যেই। এটাই তো দরকার □ যাই হোক ফাংশনের ম্যাথেমেটিক্যাল রিপ্রেজেন্টেশন এবং গ্রাফটা দেখতে নিচের মত

$$\frac{1}{1+e^{-x}}$$



তো আমরা আমাদের ওয়েটেড সাম কে এই ফাংশনের সাহায্যে 1 ও 0 এর মধ্যে এনে ফেলতে পারি। Sigmoid Function নিয়ে পড়ার জায়গা এটা না। দরকার হলে আলাদা করে দেখে ফিরে আসতে পারেন এই পোস্টে।

দ্বিতীয় ধাপে – এরর এর পরিমাণ বের করবো। এটা খুবি সহজ কাজ। প্রত্যেকটি ইনপুট কম্বিনেশনের জন্য আউটপুট থেকে এই নিউরাল নেটওয়ার্কের হিসেব করা আউটপুটকে বিয়োগ দিতে হবে।

তৃতীয় ধাপে – আমরা এররের উপর ভিত্তি করে ওয়েট অ্যাডজাস্ট করবো। কিন্তু কি পরিমাণে অ্যাডজাস্ট করবো? এক্ষেত্রেও আমরা একটা ফর্মুলা “Error Weighted Derivative” ব্যবহার করতে পারি। সূত্রটা দেখতে নিচের মতোঃ

$$\text{Adjust weights by} = error \cdot input \cdot SigmoidCurveGradient(output)$$

সূত্রে বিভিন্ন ফ্যাক্টরের প্রয়োজনীয়তা নিয়ে প্রশ্ন আসতেই পারে। বিষয়টা সহজ – যেহেতু আমরা এরর সমানুপাতে ওয়েট অ্যাড জাস্ট করবো তাই এটাকে লাগছে। আবার সাথে আমরা ইনপুটকেও নিচ্ছি যাতে করে ইনপুট যদি 0 হয় তাহলে ওয়েট অ্যাডজাস্ট করবো না (ডান পাশে শুন্য আসবে)। আসলে ইনপুট শুন্য হলে অ্যাডজাস্ট করে লাভও নাই। ওই Edge এর মাণ এমনিতেই শূন্য আসবে। তিন নাম্বার ফ্যাক্টরটা গুরুত্বপূর্ণ যা কিনা একটি ভ্যালুর জন্য (এক্ষেত্রে আমাদের ক্যালকুলেট করা আউটপুট) প্রাপ্ত Sigmoid Curve এর Gradient (ঢাল)। সূত্রের ডান পাশে এই ফ্যাক্টর এর তাৎপর্য এরকম – Sigmoid Curve এর মাধ্যমে আমরা নিউরনের আউপুট হিসেবে করছি। যদি এই আউটপুটের মাণ খুব বেশি পজিটিভ বা খুব বেশি নেগেটিভ হয় তার মাণে নিউরনটি ট্রেনিং সেটের আউটপুট ভ্যালুর দিকে ঝুঁকতে খুবি আত্মবিশ্বাসী তথা ওয়েট অ্যাডজাস্ট করার খুব একটা দরকার নাই। অন্যদিকে আমরা Sigmoid Function এর গ্রাফ থেকে দেখতে পারি যে – ভ্যালু যত বেশি, এই Curve এর Gradient বা ঢাল তত কম। তাই এই ফ্যাক্টরকে ডান পাশে রাখলে এবং আউটপুট খুব বেশি পজিটিভ/নেগেটিভ আসলে এই ফ্যাক্টরের মাণও কম আসতেছে আর তাই বাম পাশে অ্যাডজাস্টের পরিমাণও কম হচ্ছে। বুদ্ধি □

যাই হোক Sigmoid Curve এর Gradient বের করার সহজ সূত্র হচ্ছেঃ

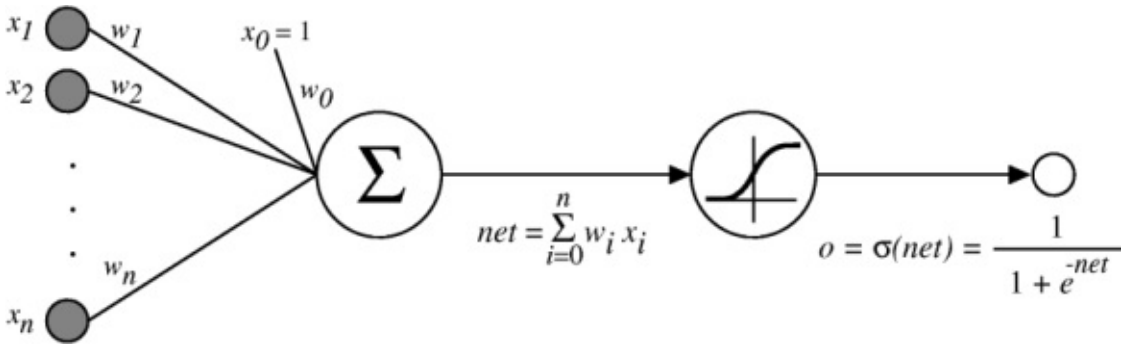
$$\mathrm{SigmoidCurveGradient(output)} = output \cdot (1 - output)$$

অর্থাৎ Adjust weight by সমীকরণ দাড়ায়ঃ

$$\mathrm{Adjust\ weights\ by} = error \cdot input \cdot output \cdot (1 - output)$$

ফিরে দেখা

এ অবস্থায় আমরা আমাদের সিঙ্গেল নিউরন বিশিষ্ট আর্টিফিশিয়াল নিউরাল নেটওয়ার্কের চেহারাটা একটু স্মার্ট ভাবে দেখি,



এখানে n এর মান 3 ধরলেই আমাদের সমস্যার সাথে উপরের ফিগারটি মিলে যাবে।

“Talk is cheap. Show me the code.”

এই বানীটি কার, তার নাম জানলে নিচে কমেন্ট করতে পারেন। এই যে আমরা নিউরাল নেটওয়ার্ক এর কাহিনীকে ফলো করে সেরকম নীতিতে আমাদের বাস্তবের কিছু সমস্যা সমাধানের জন্য একটা পদ্ধতি নিয়ে চিন্তা করলাম সেটা তো আর খাতা কলমে করে কুলাবে না। এই কাজটা কম্পিউটার দিয়ে করালে খুব দ্রুত আমাদের উদ্দেশ্য পুড়ন হবে। আর সবাই জানে, কম্পিউটারকে দিয়ে ইচ্ছামত কামলা খাটুনি খাটিয়ে নেয়া যায়। শুধুমাত্র তাকে তার ভাষায় আদেশ দিতে হবে। এর নাম নাকি আবার কম্পিউটার প্রোগ্রাম। তো, কি আর করা, লিখে ফেলি; কম্পিউটার বোঝে এবং আমাদের লিখতে সহজ এমন একটা ভাষায় একটা প্রোগ্রাম, যার মাধ্যমে বস্তুতপক্ষে আমরা উপড়ে আলোচ্য কাজ গুলোকেই করবো।

যদি আপনার পাইথন প্রোগ্রামিং ল্যাঙ্গুয়েজে ভালো দখল থাকে তাহলে আপনার জন্য ডাটা সায়েন্স, মেশিন লার্নিং এবং ডিপ লার্নিং নিয়ে কাজ করা সহজ হয়ে যায়। আমরা নিচে একটা পূর্ণ প্রোগ্রাম দেখবো যার মাধ্যমে তিনটি ইনপুট ওয়ালা একটি সিঙ্গেল নিউরন তৈরি করা হয়েছে এবং সেই ইনপুট এইজ গুলোতে প্রথমে কিছু র‍্যান্ডোম ওয়েট সেট করা হয়েছে। এরপর ওই নিউরনে ট্রেনিং ডাটাসেট অর্থাৎ কিছু ইনপুট row এবং row সাপেক্ষে একটি করে আউটপুট দিয়ে দেয়া হয়েছে। Sigmoid Function ব্যবহার করে নিউরনের চিন্তা অনুযায়ী আউটপুট বের করা হয়েছে। সত্যিকারের আউপুট এবং নিউরনের হিসাব করে বের করা আউটপুটের তুলনা করে এরর চেক করা হয়েছে। ১০০০০ বার লুপ চালিয়ে (ট্রেনিং করিয়ে) উপড়ে আলোচ্য ওয়েট অ্যাডজাস্ট করার সূত্র দিয়ে প্রত্যেক লুপের মধ্যে একবার করে ওয়েট অ্যাডজাস্ট করা হয়েছে। সবশেষে একই নিউরনে নতুন একটি ডাটাসেট দিয়ে তার আউপুট জানতে চাওয়া হয়েছে। যদি সে আমাদের ধারনা করা আউপুটকেই আউটপুট হিসেবে দিতে পারে তাহলে বলা যায় যে, এই সিঙ্গেল নিউরন ওয়ালা নেটওয়ার্কটি ৪টি ট্রেনিং ডাটাসেট থেকেই প্যাটার্ন খুঁজে নিতে সফল হয়েছে এবং সেই প্যাটার্ন মোতাবেক নতুন ডাটা সেটের জন্য আউটপুট বলে দিতে পারছে।

Medium কমিউনিটির ব্লগার @miloharper এর gist থেকে ফর্ক করা প্রোগ্রামটি নিচে দেয়া হলঃ

```
from numpy import exp, array, random, dot

class NeuralNetwork():
    def __init__(self):
        # Seed the random number generator, so it generates the same numbers
        # every time the program runs.
        random.seed(1)

        # We model a single neuron, with 3 input connections and 1 output connection.
        # We assign random weights to a 3 x 1 matrix, with values in the range -1 to 1
        # and mean 0.
        self.synaptic_weights = 2 * random.random((3, 1)) - 1

    # The Sigmoid function, which describes an S shaped curve.
    # We pass the weighted sum of the inputs through this function to
    # normalise them between 0 and 1.
    def __sigmoid(self, x):
        return 1 / (1 + exp(-x))
```

```
    # The derivative of the Sigmoid function.
    # This is the gradient of the Sigmoid curve.
    # It indicates how confident we are about the existing weight.
    def __sigmoid_derivative(self, x):
        return x * (1 - x)

    # We train the neural network through a process of trial and error.
    # Adjusting the synaptic weights each time.
    def train(self, training_set_inputs, training_set_outputs, number_of_training_iter
ations):
        for iteration in range(number_of_training_iterations):
            # Pass the training set through our neural network (a single neuron).
            output = self.think(training_set_inputs)
            # print("\nOutput of the Above Function After Sigmoid Applied: \n",output)

            # Calculate the error (The difference between the desired output
            # and the predicted output).
            error = training_set_outputs - output
            # print("\nTraining Set Output Matrix: \n", training_set_outputs)
            # print("\nError: Training Set Output Matrix 4x1 - Above Matrix 4x1 \n", e
rror)

            # Multiply the error by the input and again by the gradient of the Sigmoid
 curve.
            # This means less confident weights are adjusted more.
            # This means inputs, which are zero, do not cause changes to the weights.
            adjustment = dot(training_set_inputs.T, error * self.__sigmoid_derivative(
output))
            # print("\nAdjustment Matrix: \n", adjustment)

            # Adjust the weights.
            self.synaptic_weights += adjustment

    # The neural network thinks.
    def think(self, inputs):
        dot_product = dot(inputs, self.synaptic_weights)
        # print("\nDot Product of Input Matrix and Weight Matrix: \n",dot_product)
        # Pass inputs through our neural network (our single neuron).
        return self.__sigmoid(dot_product)

if __name__ == "__main__":

    #Intialise a single neuron neural network.
    neural_network = NeuralNetwork()

    print ("\n\nRandom starting synaptic weights: ")
    print (neural_network.synaptic_weights)

    # The training set. We have 4 examples, each consisting of 3 input values
    # and 1 output value.
    training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
    training_set_outputs = array([[0, 1, 1, 0]]).T
```

```
    # Train the neural network using a training set.
    # Do it 10,000 times and make small adjustments each time.
    neural_network.train(training_set_inputs, training_set_outputs, 10000)

    print ("\nNew synaptic weights after training: ")
    print (neural_network.synaptic_weights)

    # Test the neural network with a new situation.
    print ("\nConsidering new situation [1, 0, 0] -&amp;amp;amp;amp;amp;amp;amp;amp;amp;am
p;gt; ?: ")
    print (neural_network.think(array([1, 0, 0])))
```

যদি প্রোগ্রামটির `if __name__ == "__main__":` থেকে দেখা শুরু করেন তাহলে ধাপে ধাপে বুঝতে পারার কথা কিভাবে কোডের মাধ্যমে এই নিউরাল নেটওয়ার্ক তৈরি করা হয়েছে। আমি যথা সম্ভব আরেক্টু সহজে ব্যাখ্যা করার চেষ্টা করছি। প্রোগ্রামের শুরুতেই numpy লাইব্রেরী যুক্ত করা হয়েছে যাতে করে খুব সহজে ম্যাট্রিক্স পদ্ধতিতে কিছু ক্যালকুলেশনের কাজ করা যায় কারণ ন্যাটিভ পাইথনে ম্যাট্রিক্স টাইপের কোন ডাটা স্ট্রাকচার নাই। অন্যদিকে নিউরাল নেটওয়ার্কের গঠন মোতাবেক ইনপুট এবং ওয়েট নিয়ে গুন/যোগ ইত্যাদি করার সময় ম্যাট্রিক্স স্টাইল ভালো উপায়।

যেমন, এর মাধ্যমে আমাদের ট্রেনিং ডাটাসেট গুলোকে খুব সহজে ম্যাট্রিক্স এর রূপ দিতে পারি নিচের মত করে।

```
training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
training_set_outputs = array([[0, 1, 1, 0]]).T
```

উল্লেখ্য – আউপুট কলামের ডাটা গুলোকে প্রথমে একটি 1×4 সাইজের ম্যাট্রিক্সে স্টোর করে তারপর ট্রান্সপোজ করে 4×1 সাইজে কনভার্ট করা হয়েছে যাতে ভিজুয়াল রিপ্রেজেন্টেশন মনে করা যেতে পারে এমন –

ইনপুট ম্যাট্রিক্স –

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

আউটপুট ম্যাট্রিক্স –

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

এভাবে ডাটা স্টোর করার ফলে আমরা একবারে পুরো ইনপুট ডাটা টেবিলকে আমাদের নিউরাল নেটওয়ার্কে ইনপুট দিয়ে খুব সহজে প্রত্যেকটা ইনপুট সেট (এ ক্ষেত্রে এক একটা row) -এর সাথে ওয়েট সেট ডট গুন করে একবারেই একটা আউটপুট ম্যাট্রিক্স পেয়ে যেতে পারি যেখানে ৪টা ইনপুট সেটের (৪টা row) জন্যই ৪টা আউটপুট ভ্যালু থাকবে 4×1 সাইজে। এতে করে প্রত্যেকটা Epoch এ পুরো অপারেশনটা একবার পুরোপুরি শেষ হবে। এছাড়াও এই লাইব্রেরী থেকে আরও কিছু ফাংশনের সাহায্য নিয়ে কিছু অপারেশনকে সহজ বোধ্য করা হয়েছে।

স্ক্রিপ্ট হিসেবে এই প্রোগ্রামকে রান করালে ৫৮ নাম্বার লাইনে থেকেই এই প্রোগ্রামটির কার্যক্রম শুরু হয়। শুরুতেই NeuralNetwork ক্লাসের একটি অবজেক্ট তৈরি করা হয়েছে যার মাধ্যমে ফ্রেশ একটি নিউরাল নেটওয়ার্ক তৈরি করা যায়। তো, দেখে আসি সেই ক্লাসের চেহারাটা। ৪নাম্বার লাইনে ক্লাসকে ডিফাইন করা হয়েছে। এর কন্সট্রাক্টরের মধ্যেই আমাদের সেই বহুল আলোচিত র‍্যান্ডম ওয়েট তিনটি তৈরি করা হচ্ছে।

যেহেতু আমাদের নিউরনের ৩টি ইনপুট তাই তিনটি ইনপুটের জন্য তিনটি ওয়েট নির্ধারণ করে ইনপুট গুলোর সাথে গুন করতে কাজ করার সুবিধার্থে 3×1 সাইজের একটি ম্যাট্রিক্স নেয়া/তৈরি করা হয়েছে synaptic_weights নামে। প্রথমবার অর্থাৎ ওয়েট অ্যাডজাস্ট হবার আগে এর চেহারা হতে পারে এমন –

$$\begin{bmatrix} -0.16595599 \\ 0.44064899 \\ -0.99977125 \end{bmatrix}$$

নোটঃ আপনি প্রোগ্রাম রান করানোর সময় আলাদা ভ্যালু পেতে পারেন কারন র‍্যান্ডমলি জেনারেটেড।

এই ক্লাসের মধ্যে আরেকটি মেথড বানানো হয়েছে যার মাধ্যমে Sigmoid Function ব্যবহার করে ভ্যালু নরমালাইজেশন অর্থাৎ আউটপুট ভ্যালুকে 1 ও 0 মাঝে রাখা হয়। তার নিচেই আছে আরেকটি ফাংশন যার মাধ্যমে আমরা যেকোনো একটি আউটপুট ভ্যালুর জন্য Sigmoid Curve এর Gradient বের করতে পারি। এটি কাজে লাগে ওয়েট অ্যাডজাস্টমেন্ট এর মান ঠিক করতে। উপড়ে এটা নিয়ে আলোচনা করা হয়েছে। এরপরেই আছে গুরুত্বপূর্ণ train ফাংশন যার মাধ্যমে আমাদের নিউরাল নেটওয়ার্কটি প্যাটার্ন চেনা শিখে নেয়।

প্রথমেই একটি লুপ চালানো হয়েছে যার মাণ নির্ধারণ করবে আপনি যতগুলো Epoch বা ট্রেনিং সাইকেল করাতে চান তার উপর। এখানে ১০০০০ বার Forward এবং Back Propogaion করাতে বলা হচ্ছে। ১০০০০ লুপের প্রথম iteration -এ লুপের মধ্যের প্রথম কাজ হচ্ছে think ফাংশনের ব্যবহার করে এবং র‍্যান্ডম ওয়েটের উপর ভিত্তি করে একটা আউটপুট ম্যাট্রিক্স তৈরি করা যার মধ্যে নিউরনের হিসাব মোতাবেক পাওয়া আউটপুট গুলো থাকবে। এটির ডাইমেনশন হবে 4×1 অর্থাৎ ৪সেট ইনপুট ডাটার (৪টি row) জন্য ৪টি আউটপুট তথা নিচের মত একটি ম্যাট্রিক্স।

$$\begin{bmatrix} -0.99977125 \\ -0.72507825 \\ -1.16572724 \\ -0.55912226 \end{bmatrix}$$

যদি think ফাংশনের কোড দেখি তাহলে দেখতে পারবো যে এখানে 4×3 সাইজের পুরো ইনপুট ডাটা টেবিল যাকে ম্যাট্রিক্সে কনভার্ট করা হয়েছে, তার সাথে 3×1 সাইজের ওয়েট ম্যাট্রিক্সের গুন করা হয়েছে। এতে করে বস্তুত প্রত্যেকটি ইনপুট সেট যেমন প্রথমত 0 0 1 এর সাথে তিনটি ওয়েট

$$\begin{bmatrix} -0.16595599 \\ 0.44064899 \\ -0.99977125 \end{bmatrix}$$

– কে ডট গুন করা হয়েছে। আবার দ্বিতীয় ইনপুট সেট 1 1 1 এর সাথে একই ওয়েট ম্যাট্রিক্স

$$\begin{bmatrix} -0.16595599 \\ 0.44064899 \\ -0.99977125 \end{bmatrix}$$

– কে ডট গুন করা হয়েছে। অর্থাৎ এভাবে সব গুলো ইনপুট কম্বিনেশনের সাথেই একবার করে ওই তিনটি ওয়েট ডট গুন করা হয়েছে। এভাবে যে আউটপুট ম্যাট্রিক্স পাওয়া যায় সেটাও কিন্তু 4×1 সাইজের ম্যাট্রিক্স। সেই ম্যাট্রিক্সকে একবার করে __sigmoid মধ্যে চালিয়ে নিয়ে ভ্যালু গুলোকে নরমালাইজ করা হয়েছে। তো, সব গুলো ইনপুট কম্বিনেশন এর সাথে ওয়েট গুলোর ডট গুন (গুন ও গুন গুলোর যোগ) করে নরমালাইজ করার পর নিচের মত একটি ম্যাট্রিক্স পাওয়া যাবে,

$$\begin{bmatrix} 0.2689864 \\ 0.3262757 \\ 0.23762817 \\ 0.36375058 \end{bmatrix}$$

এই ম্যাট্রিক্সকে output ভ্যারিয়েবলে স্টোর করা হচ্ছে। এরপর এরর হিসাবের জন্য আমরা 4×1 সাইজের ট্রেনিং আউটপুট ম্যাট্রিক্স তথা,

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

থেকে উপরের 4×1 সাইজের output ম্যাট্রিক্স বিয়োগ করে নিচের মত একটি ম্যাট্রিক্স পেতে পারি,

$$\begin{bmatrix} -0.2689864 \\ 0.6737243 \\ 0.76237183 \\ -0.36375058 \end{bmatrix}$$

এরপর এই এরর ম্যাট্রিক্স কে সাথে নিয়ে ইনপুট ডাটা সেট ম্যাট্রিক্স এবং Sigmoid Derivative কে কাজে লাগিয়ে অ্যাডজাস্টমেন্ট এর পরিমাণ বের করা হচ্ছে। এই অ্যাডজাস্টমেন্ট ম্যাট্রিক্সটিও ওয়েট ম্যাট্রিক্স এর মত 3×1 সাইজের। আর তাই train ফাংশনের শেষ লাইনে মুল ওয়েট ম্যাট্রিক্স এর সাথে এই অ্যাডজাস্ট ম্যাট্রিক্স যোগ করে ওয়েট ম্যাট্রিক্সে পরিবর্তন করে নেয়া হচ্ছে।

NeuralNetwork ক্লাসের কোড বোঝার পর আবারও ফিরে আসি পাইথন প্রোগ্রামের রেগুলার এক্সিকিউশন স্টেজে। ক্লাস ইনিসিয়ালাইজ করার পর পর্যবেক্ষণের স্বার্থে প্রথমবার সেট হওয়া র‍্যান্ডম ওয়েট ম্যাট্রিক্সকে প্রিন্ট করে দেখা হচ্ছে ওয়েট গুলো কি কি –

$$\begin{bmatrix} -0.16595599 \\ 0.44064899 \\ -0.99977125 \end{bmatrix}$$

এরপর আমাদের ডাটা টেবিল থেকে ইনপুট এবং আউটপুট গুলোকে গুছিয়ে 4×3 সাইজের ট্রেনিং সেট ইনপুট এবং 4×1 সাইজের ট্রেনিং সেট আউটপুট ম্যাট্রিক্স বানিয়ে নেয়া হচ্ছে। এরপরেই উপড়ে আলোচ্য NeuralNetwork ক্লাসের অবজেক্ট neural_network –র মেথড, train এর মধ্যে এগুলো পাঠিয়ে দেয়া হচ্ছে। ১০০০০ বার চক্কর দেয়ার পর অপ্টিমাইজ ওয়েট ম্যাট্রিক্সটি কেমন রূপ ধারণ করলো সেটাও প্রিন্ট করা হচ্ছে।

$$\begin{bmatrix} 9.67299303 \\ -0.2078435 \\ -4.62963669 \end{bmatrix}$$

সবশেষে, একটি নতুন ইনপুট সেট কে think ফাংশনে পাঠিয়ে আমাদের নিউরাল নেটওয়ার্ক এর কাছে আউটপুট জানতে চাওয়া হচ্ছে। এবার think ফাংশন, এই ইনপুট ডাটা সেট তথা 1×3 ম্যাট্রিক্সের সাথে আপডেটেড 3×1 ওয়েট ম্যাট্রিক্স এর ডট গুন করে Sigmoid অ্যাপ্লাই করে নরমালাইজ ডাটা তথা 1 থেকে 0 মধ্যের একটা ভ্যালুকে প্রিন্ট করে 1×1 সাইজের ম্যাট্রিক্স আকারে যেটা কিনা আমাদের নিউরাল নেটওয়ার্কের প্রেডিকশন।

আর সেটি হচ্ছে,

$$\begin{bmatrix} 0.99993704 \end{bmatrix}$$

অর্থাৎ আমাদের নিউরাল নেটওয়ার্ক ভালোমতই ইনপুট ডাটা থেকে প্যাটার্ন খুঁজে তার উপর ভিত্তি করে পরবর্তী নতুন ইনপুট ডাটার জন্য তার আউটপুট কি হবে সেটা বলে দিতে পারছে।

আপনি যদি প্রথম iteration এর সব গুলো কাজের ধাপকে লগ করে দেখতে চান যে একটা ট্রেনিং লুপে কি কি ঘটছে তাহলে ৭৩ নাম্বার লাইনে 10000 এর পরিবর্তে 1 পাঠিয়ে এবং পুরো প্রোগ্রামের মধ্যে থাকা কমেন্ট করা প্রিন্ট স্টেটমেন্ট গুলোকে অ্যাক্টিভ করে দেখতে পারেন নিচের মত আউটপুট এবং সেগুলো ম্যানুয়ালি বিচার করতে পারেন।

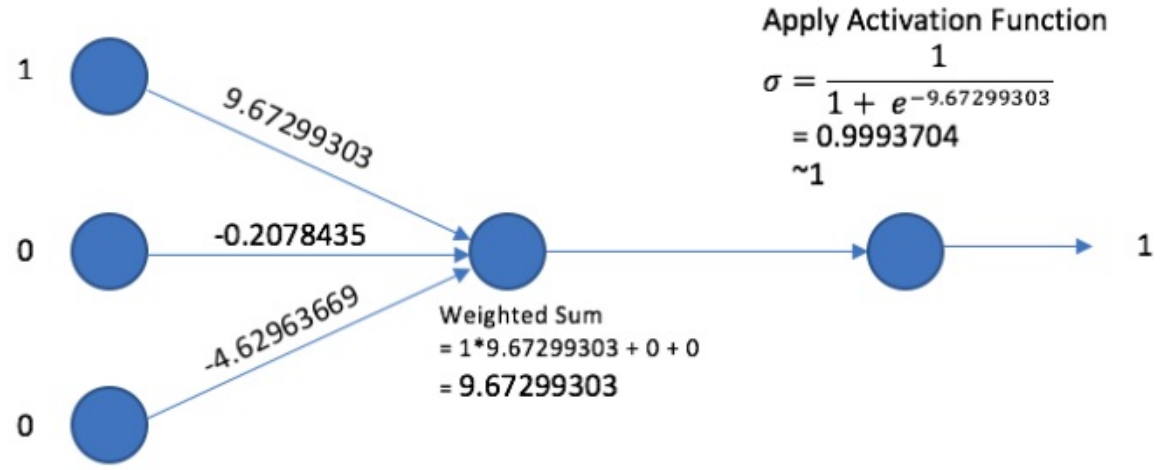
```
Random starting synaptic weights:
[[-0.16595599]
 [ 0.44064899]
 [-0.99977125]]

Dot Product of Input Matrix and Weight Matrix:
 [[-0.99977125]
 [-0.72507825]
 [-1.16572724]
 [-0.55912226]]

Output of the Above Function After Sigmoid Applied:
 [[ 0.2689864 ]
 [ 0.3262757 ]
 [ 0.23762817]
 [ 0.36375058]]

Training Set Output Matrix:
 [[0]
 [1]
 [1]
 [0]]

Error: Training Set Output Matrix 4x1 - Above Matrix 4x1
 [[-0.2689864 ]
 [ 0.6737243 ]
 [ 0.76237183]
 [-0.36375058]]

Adjustment Matrix:
 [[ 0.28621005]
 [ 0.06391297]
 [ 0.14913351]]

New synaptic weights after training:
[[ 0.12025406]
 [ 0.50456196]
 [-0.85063774]]

Considering new situation [1, 0, 0] -> ?:
```

পুরো ১০০০০ বার লুপের পর অর্থাৎ ট্রেনিং শেষের পর ওয়েটেড ম্যাট্রিক্স এর ফাইনাল রূপ আসবে নিচের মত,

$$\begin{bmatrix} 9.67299303 \\ -0.2078435 \\ -4.62963669 \end{bmatrix}$$

পরীক্ষা করে দেখা

তাহলে চলুন এবার খাতা কলম নিয়ে পরীক্ষায় বসে যাই কিভাবে এই কৃত্রিম নিউরনের নেটওয়ার্ক এই সিদ্ধান্ত নিতে পারলো। আপডেটেড ওয়েট গুলোকে এবার নিউরাল নেটওয়ার্কের ফিগারে বসাই,



অভিনন্দন আপনি একদম ব্যাসিক একটা নিউরাল নেটওয়ার্ক তৈরি করা শিখেছেন এবং এর ব্যাসিক কার্যনীতি বুঝে ফেলেছেন। আমাদের এই নিউরাল নেটওয়ার্কে একটা ইনপুট লেয়ার এবং একটা আউটপুট লেয়ার। সামনের পোস্টে নিচের সমস্যাটি সমাধান করা হবে যা কিনা এরকম ব্যাসিক নেটওয়ার্ক দিয়ে সমাধান করা সম্ভব নয় অর্থাৎ এরকম একটি নেটওয়ার্ক নিচের সমস্যা থেকে প্যাটার্ন খুঁজে পেতে অপারগ। এর জন্য ডিপ নিউরাল নেটওয়ার্ক ডিজাইন করবো পরবর্তী পোস্টে। ততক্ষণ পর্যন্ত আপনি বোঝার চেষ্টা করুন তো কি প্যাটার্ন লুকিয়ে আছে এই সমস্যাটিতে এবং আউটপুট কি হতে পারে?

	ইনপুট কম্বিনেশন			আউটপুট
উদাহরণ ১	0	0	1	0
উদাহরণ ২	0	1	1	1
উদাহরণ ৩	1	0	1	1
উদাহরণ ৪	0	1	0	1
উদাহরণ ৫	1	0	0	1
উদাহরণ ৬	1	1	1	0
উদাহরণ ৭	0	0	0	0

নতুন অচেনা ইনপুট কম্বিনেশন	1	1	0	?

মাল্টি লেয়ার অর্থাৎ ডিপ নিউরাল নেটওয়ার্ক

আগের চ্যাপ্টারে উল্লেখ করা সমস্যাটি ছিল,

	ইনপুট কম্বিনেশন			আউটপুট
উদাহরণ ১	0	0	1	0
উদাহরণ ২	0	1	1	1
উদাহরণ ৩	1	0	1	1
উদাহরণ ৪	0	1	0	1
উদাহরণ ৫	1	0	0	1
উদাহরণ ৬	1	1	1	0
উদাহরণ ৭	0	0	0	0

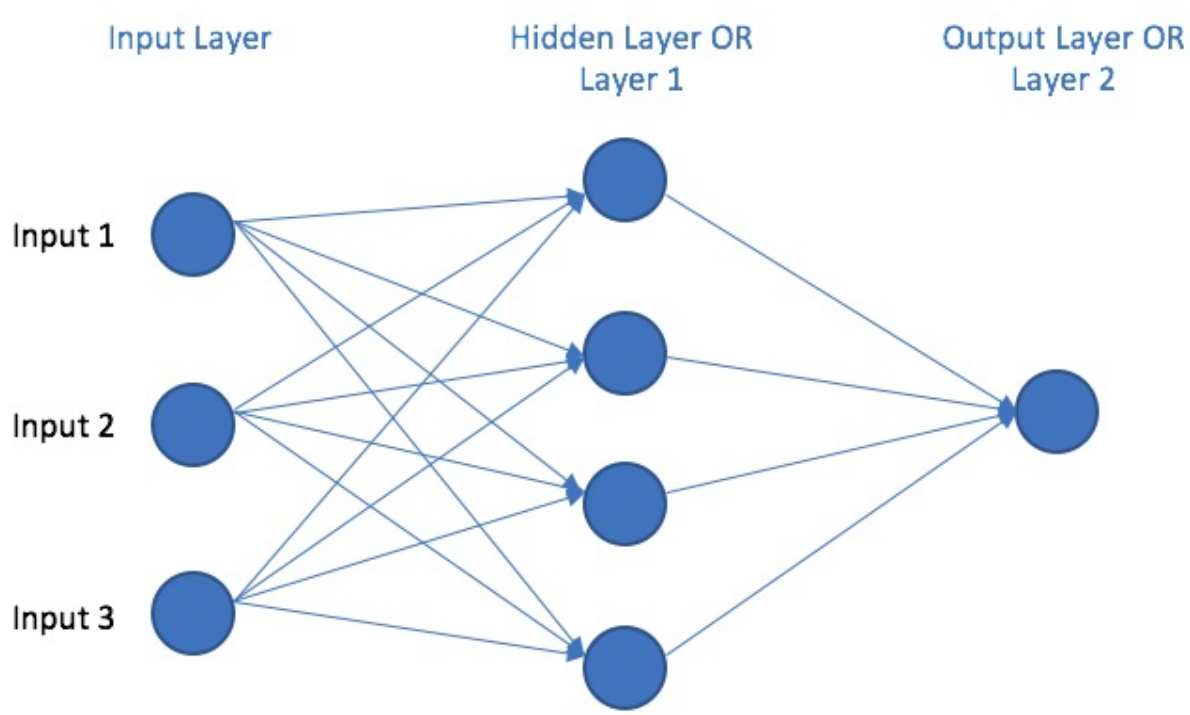
নতুন অচেনা ইনপুট কম্বিনেশন	1	1	0	?

জটিল প্যাটার্ন খুঁজে নিতে যেমন একাধিক লেয়ার এবং নিউরনের সংখ্যা বেশি লাগবে তেমনি লাগবে বেশি পরিমাণ ট্রেনিং ডাটা। আমরা নিজেরা যেমন, কোন প্যাটার্ন বুঝতে গিয়ে প্রশ্নকর্তাকে জিজ্ঞেস করি যে আরও কয়েকটা উদাহরণ দাও, তেমনি নিউরাল নেটওয়ার্কও জটিল এবং কনফিউজিং প্যাটার্ন বুঝতে গিয়ে যত বেশি উদাহরণ পাবে তত সঠিকভাবে প্যাটার্ন চিনতে পারবে।

এখানে প্যাটার্নটা হচ্ছে এরকম - ইনপুট কম্বিনেশনের তৃতীয় কলামের ভ্যালু অনর্থক এবং প্রথম দুই কলামের মধ্যে XOR অপারেশনের উপর ভিত্তি করে আউটপুট নির্ধারীত হচ্ছে। আর তাই, 1 1 0 এর আউটপুট হবে 1 XOR 1 = 0.</p>

এই ধরনের প্যাটার্নকে Non Linear প্যাটার্ন বলা হয়ে থাকে। কারণ এখানে ইনপুট এবং আউটপুটের মধ্যে সরাসরি কোন one-to-one রিলেশন নাই।তাই এই প্যাটার্নকে উদ্ধার করার ক্ষমতা আমাদের আগের সিঙ্গেল নিউরন নেটওয়ার্কের নাই। বরং আমাদের একটি হিডেন লেয়ার ওয়ালা ডিপ নিউরাল নেটওয়ার্ক ডিজাইন করতে হবে।

এই নতুন লেয়ারে ৪টি নিউরন থাকতে পারে যেগুলো এই নিউরাল নেটওয়ার্ককে ইনপুট কম্বিনেশন গুলো নিয়ে একটু অন্যভাবে চিন্তা করাতে সাহায্য করে। চিন্তা কি জিনিষ আগেই একবার বলা হয়ে গেছে।



উপরের ডায়াগ্রাম থেকে দেখা যাচ্ছে যে, Layer 1 এর আউটপুট গুলো Layer 2 এর ইনপুট হিসেবে যাচ্ছে। এভাবে আমাদের নিউরাল নেটওয়ার্ক, লেয়ার ১ এর আউটপুট এর সাথে ট্রেনিং সেট আউটপুটেরও একটা কো-রিলেশন বের করতে পারবে। নিউরনের লার্নিং এর সাথে সাথে এই দুই লেয়ারের ওয়েট অ্যাডজাস্ট করে করে এই কো-রিলেশন বাড়তে থাকবে।

বলে নেয়া ভালো, এই বিষয়টার সাথে ইমেজ রিকগনিশনের টেকনিকের মিল আছে। অর্থাৎ যদি আমরা একটি আপেলের ফটোর কথা চিন্তা করি, সেখানে কিন্তু প্রত্যেকটা পিক্সেল (ভ্যালু) এর সাথে বস্তুত আপেলের কোন সম্পর্ক নাই। দুইটা দুই জগতের জিনিষ। কিন্তু আবার [কিছু পিক্সেল কম্বিনেশন] এবং [আপেল] এই দুটো ফ্যাক্টরের রিলেশনশিপ আছে। অর্থাৎ উপরের নেটওয়ার্কে, প্রথম raw input এর সাথে আউটপুট এর সরাসরি কোন সম্পর্ক নাই (এটা আমরা জানি, ধরে নিচ্ছি) কিন্তু লেয়ার ১ এর আউটপুট তথা পিছনের কম্বিনেশনের সাথে মুল ডাটা সেটের একটা রিলেশন থাকতে পারে। আর তাই এখানে মধ্যবর্তী লেয়ারের আবির্ভাব এবং প্রয়োজনীয়তা।

> এই যে, বিভিন্ন স্টেজের মধ্যেকার কো-রিলেশনকে চেনার জন্য এবং কাজে লাগানোর জন্য এক বা একাধিক মধ্যবর্তী লেয়ারের সংযোজন, এটাকেই ডিপ লার্নিং বলে।

আবারও প্রোগ্রামিং

এখন আমরা আগের চ্যাপ্টারে ডিজাইন করা ডিপ নিউরাল নেটওয়ার্কের একটা প্রোগ্রাম্যাটিক ভার্সন দেখবোঃ

```
from numpy import exp, array, random, dot

class NeuronLayer():
    def __init__(self, number_of_neurons, number_of_inputs_per_neuron):
        self.synaptic_weights = 2 * random.random((number_of_inputs_per_neuron, number
_of_neurons)) - 1

class NeuralNetwork():
    def __init__(self, layer1, layer2):
        self.layer1 = layer1
        self.layer2 = layer2

    # The Sigmoid function, which describes an S shaped curve.
    # We pass the weighted sum of the inputs through this function to
    # normalise them between 0 and 1.
    def __sigmoid(self, x):
        return 1 / (1 + exp(-x))

    # The derivative of the Sigmoid function.
    # This is the gradient of the Sigmoid curve.
    # It indicates how confident we are about the existing weight.
    def __sigmoid_derivative(self, x):
        return x * (1 - x)

    # We train the neural network through a process of trial and error.
    # Adjusting the synaptic weights each time.
    def train(self, training_set_inputs, training_set_outputs, number_of_training_iter
ations):
        for iteration in range(number_of_training_iterations):
            # Pass the training set through our neural network
            output_from_layer_1, output_from_layer_2 = self.think(training_set_inputs)

            # Calculate the error for layer 2 (The difference between the desired outp
ut
            # and the predicted output).
            layer2_error = training_set_outputs - output_from_layer_2
            layer2_delta = layer2_error * self.__sigmoid_derivative(output_from_layer_
2)

            # Calculate the error for layer 1 (By looking at the weights in layer 1,
            # we can determine by how much layer 1 contributed to the error in layer 2
).
            layer1_error = layer2_delta.dot(self.layer2.synaptic_weights.T)
            layer1_delta = layer1_error * self.__sigmoid_derivative(output_from_layer_
1)
```

```
            # Calculate how much to adjust the weights by
            layer1_adjustment = training_set_inputs.T.dot(layer1_delta)
            layer2_adjustment = output_from_layer_1.T.dot(layer2_delta)

            # Adjust the weights.
            self.layer1.synaptic_weights += layer1_adjustment
            self.layer2.synaptic_weights += layer2_adjustment

    # The neural network thinks.
    def think(self, inputs):
        output_from_layer1 = self.__sigmoid(dot(inputs, self.layer1.synaptic_weights))
        output_from_layer2 = self.__sigmoid(dot(output_from_layer1, self.layer2.synapt
ic_weights))
        return output_from_layer1, output_from_layer2

    # The neural network prints its weights
    def print_weights(self):
        print ("    Layer 1 (4 neurons, each with 3 inputs): ")
        print (self.layer1.synaptic_weights)
        print ("    Layer 2 (1 neuron, with 4 inputs):")
        print (self.layer2.synaptic_weights)

if __name__ == "__main__":

    #Seed the random number generator
    random.seed(1)

    # Create layer 1 (4 neurons, each with 3 inputs)
    layer1 = NeuronLayer(4, 3)

    # Create layer 2 (a single neuron with 4 inputs)
    layer2 = NeuronLayer(1, 4)

    # Combine the layers to create a neural network
    neural_network = NeuralNetwork(layer1, layer2)

    print ("Stage 1) Random starting synaptic weights: ")
    neural_network.print_weights()

    # The training set. We have 7 examples, each consisting of 3 input values
    # and 1 output value.
    training_set_inputs = array([[0, 0, 1], [0, 1, 1], [1, 0, 1], [0, 1, 0], [1, 0, 0]
, [1, 1, 1], [0, 0, 0]])
    training_set_outputs = array([[0, 1, 1, 1, 1, 0, 0]]).T

    # Train the neural network using the training set.
    # Do it 60,000 times and make small adjustments each time.
    neural_network.train(training_set_inputs, training_set_outputs, 60000)

    print ("Stage 2) New synaptic weights after training: ")
    neural_network.print_weights()
```

```
    # Test the neural network with a new situation.
    print ("Stage 3) Considering a new situation [1, 1, 0] -&amp;amp;amp;amp;amp;amp;g
t; ?: ")
    hidden_state, output = neural_network.think(array([1, 1, 0]))
    print (output)
```

আমাদের সিঙ্গেল নিউরন ওয়ালা নেটওয়ার্কের কোডের সাথে অনেক কিছুই মিল আছে এখানে। কারণ, বেশ কিছু ফাংশনালিটি সব রকম নিউরাল নেটওয়ার্কেই লাগে। উপরের প্রোগ্রামে বাড়তি কিছু কোড আছে যেমন - প্রত্যেকবার একটি করে নতুন লেয়ার নেয়ার জন্য একটি ছোট্ট ক্লাস আছে NeuronLayer নামে (লাইন 4)। এই ক্লাসের অবজেক্ট তৈরির সময় "নিউরন সংখ্যা" এবং "প্রত্যেকটা নিউরনে আগত ইনপুটের সংখ্যা" ডিফাইন করে দিলেই ওরকম একটা লেয়ার তৈরি হয়ে যাবে। আমাদের ডায়াগ্রাম অনুযায়ী যেমন Layer 1 তৈরি করা হচ্ছে এভাবে।

এরপর আছে NeuralNetwork ক্লাস (লাইন ৯) যেটা অনেকটাই আগের প্রোগ্রামের মতই। তবে গুরুত্বপূর্ণ কিছু পরিবর্তন আছে এই কোডে। যেমন - এখানে নিউরাল নেটওয়ার্ক যখন Layer 2 -এ এসে এরর হিসাব করে তখন সেটা সে Back Propogate করে একদম শুরুতে না নিয়ে বরং Layer 1 এ নিয়ে যায় এবং ওয়েট অ্যাডজাস্ট করে। Layer 2 এর এরর নির্ভর করে লেয়ার ২ এর আউটপুট এবং আসল ট্রেনিং সেট এর আউটপুটের বিয়োগের ফলের উপর (আগের মতই) । সাথে অ্যাডজাস্টমেন্ট নির্ধারণের জন্য Sigmoid Derivative (লেয়ার ২ আউটপুট এর উপর ভিত্তি করে) এবং ইনপুট হিসেবে লেয়ার ১ এর আউটপুট তো আছেই (৩৬ এবং ৪৫ নং লাইন খেয়াল করুন)।

আর Layer 1 এর এরর কিসের উপর নির্ভর করছে সেটা একটু বুঝে শুনে খেয়াল করা উচিৎ। এখানে পার্থক্যটা (Error) এমন না যে আসল আউটপুট এবং এক্সপেক্টেড আউটপুট বিয়োগ করেই এররের ধারনা পাওয়া যাবে কারণ Layer 1 এর তো কোন ব্যবহার উপযোগী আউটপুট নাই। বরং এই লেয়ার পরবর্তী লেয়ারের এররের উপর ভূমিকা রাখে। তাই এই লেয়ারের এরর ফ্যাক্টরটা বস্তুত Layer 2 এর ওয়েট এবং এরর ডেরিভেটিভ এর সমন্বয়ের অবস্থাটা। এরপর এই লেয়ারের অ্যাডজাস্টমেন্ট এর জন্য ইনপুট ফ্যাক্টর হিসেবে লাগছে মুল ইনপুট ভ্যালু গুলো, আর আউটপুট হিসেবে এই লেয়ারের আউটপুটের Sigmoid Derivative (৪১ এবং ৪৪ নং লাইন).

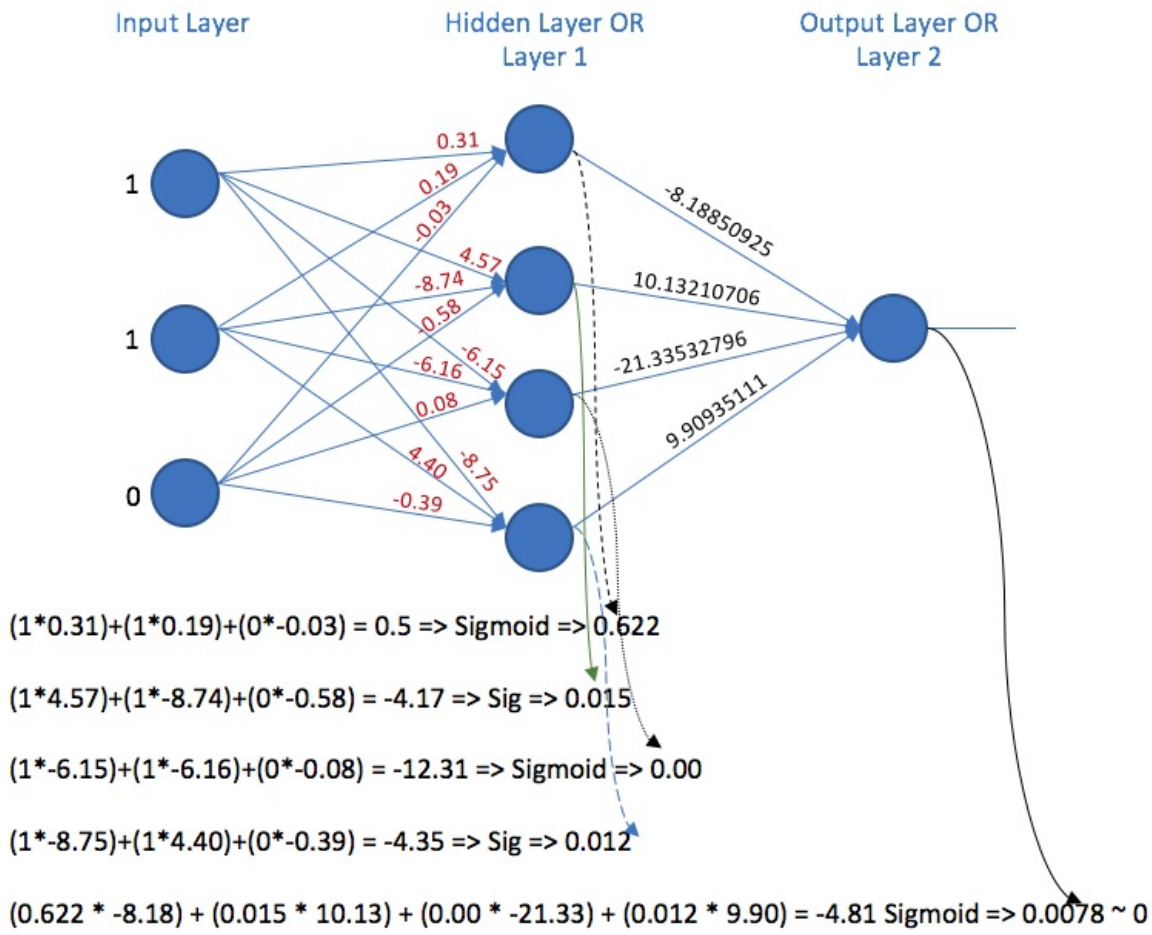
train ফাংশনের শেষে এই দুটো লেয়ারের ওয়েট গুলো অ্যাডজাস্ট করা হয়েছে এবং ট্রেনিং লুপ চালিয়ে যাওয়া হয়েছে। এর নিচে থাকা think ফাংশনের কাজ খুব সহজেই বুঝে যাওয়ার কথা কারণ এটা সেই ব্যাসিক নেটওয়ার্কের মতই (শুধু দুই ধাপের আউটপুট আলাদা করে চিন্তা করছে)।

উপরের প্রোগ্রামটি রান করালে নিচের মত আউটপুট আসতে পারেঃ

```
Stage 1) Random starting synaptic weights:
    Layer 1 (4 neurons, each with 3 inputs):
[[-0.16595599  0.44064899 -0.99977125 -0.39533485]
 [-0.70648822 -0.81532281 -0.62747958 -0.30887855]
 [-0.20646505  0.07763347 -0.16161097  0.370439  ]]
    Layer 2 (1 neuron, with 4 inputs):
[[-0.5910955 ]
 [ 0.75623487]
 [-0.94522481]
 [ 0.34093502]]
Stage 2) New synaptic weights after training:
    Layer 1 (4 neurons, each with 3 inputs):
[[ 0.3122465   4.57704063 -6.15329916 -8.75834924]
 [ 0.19676933 -8.74975548 -6.1638187   4.40720501]
 [-0.03327074 -0.58272995  0.08319184 -0.39787635]]
    Layer 2 (1 neuron, with 4 inputs):
[[ -8.18850925]
 [ 10.13210706]
 [-21.33532796]
 [  9.90935111]]
Stage 3) Considering a new situation [1, 1, 0] -> ?:
[ 0.0078876]
```

পুরো ঘটনাকে ৩টী স্টেজে ভাগ করে নিলে আমরা দেখতে পাই যে - প্রথম ধাপে শুধুমাত্র দুই লেয়ারের প্রত্যেকটি Edge এর ওয়েটকে র‍্যান্ডোমলি নির্ধারণ করা হচ্ছে। পরবর্তী ধাপে পুরো ট্রেনিং প্রসেস শেষে দুটো লেয়ারের প্রত্যেকটি এইজের আপডেটেড এবং অপ্টিমাইজড ওয়েট গুলো দেখতে পাচ্ছি। এবং তৃতীয় ধাপে নিউরাল নেটওয়ার্কে নতুন একটি অচেনা ইনপুট কম্বিনেশন দিয়ে আমরা অউটপুট পাচ্ছি 0.0078 অর্থাৎ সফলভাবে 0 প্রেডিক্ট করতে পারছে আমাদের ডিপ নিউরাল নেটওয়ার্ক :) :D

খাতা কলমে পরীক্ষা করে দেখা



নোটঃ উপরোক্ত দুটি টিউটরিয়ালের সিমপ্লিসিটির জন্য এখানে bias ফ্যাক্টরক এড়িয়ে যাওয়া হচ্ছে। bias হচ্ছে ইনপুট এবং ওয়েটের গুন ফলের সাথে আরেকটি কন্সট্যান্ট টাইপ ভ্যালু যোগ করা। অর্থাৎ যদি একটি নিউরনে একটি Edge এর মাধ্যমে ইনপুট আসে x এবং এর সাথে Edge এর ওয়েট গুন হয় w তাহলে এর সাথে আরেকটি কন্সট্যান্ট (bias) b যোগ করা যেতে পারে নিউরন বা নেটওয়ার্কের নোডে। তাহলে ওই নিউরন বা নোডে উক্ত এইজ, ওয়েটের সাপেক্ষে ভ্যালু জমা হবে z, যেখানে z = wx+b. এই z কে Activation Function এর ইনপুট হিসেবে পরে ব্যবহার করা হয়। আরেকভাবে বলা যায় - weight হচ্ছে Edge বা কানেকশনের প্রোপার্টি আর bias হচ্ছে নিউরন বা নোডের প্রোপার্টি।

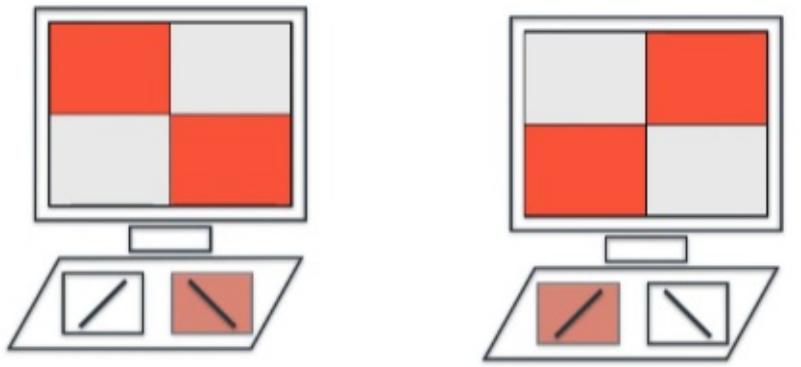
কনভলিউশনাল নিউরাল নেটওয়ার্ক

এতক্ষণ পর্যন্ত খেয়াল করেছেন যে, বার বার আমরা ম্যাট্রিক্স নিয়ে কাজ করছি। আর কাজের কাজ বলতে আমরা যা করেছি তা হচ্ছে ডাটার মধ্যে থেকে প্যাটার্ন খুঁজে বের করা এবং সেই খুঁজে পাওয়া প্যাটার্নকে কাজে লাগিয়ে পরবর্তীতে ওই রকম ডাটা সম্পর্কিত বিষয়ে প্রেডিকশন দেয়া। মজার ব্যাপার হচ্ছে, আমরা সবাই জানি – ডিজিটাল ইমেজ বা ফটো কিন্তু আর কিছুই না একটা 2D অ্যারে বা ম্যাট্রিক্স (গ্রে স্কেল বা সাদাকালো ফটোর ক্ষেত্রে)। সেই ম্যাট্রিক্সের প্রত্যেকটি সেল হচ্ছে এক একটি পিক্সেল। আর সেলের ভ্যালু হচ্ছে পিক্সেল ভ্যালু বা কালার এর মান। আবার ম্যাট্রিক্সটার (row সংখ্যা x column সংখ্যা) মানেই হচ্ছে ওই ফটোর রেজোল্যুশন। তাহলে বিষয়টা কি দাঁড়ালো?

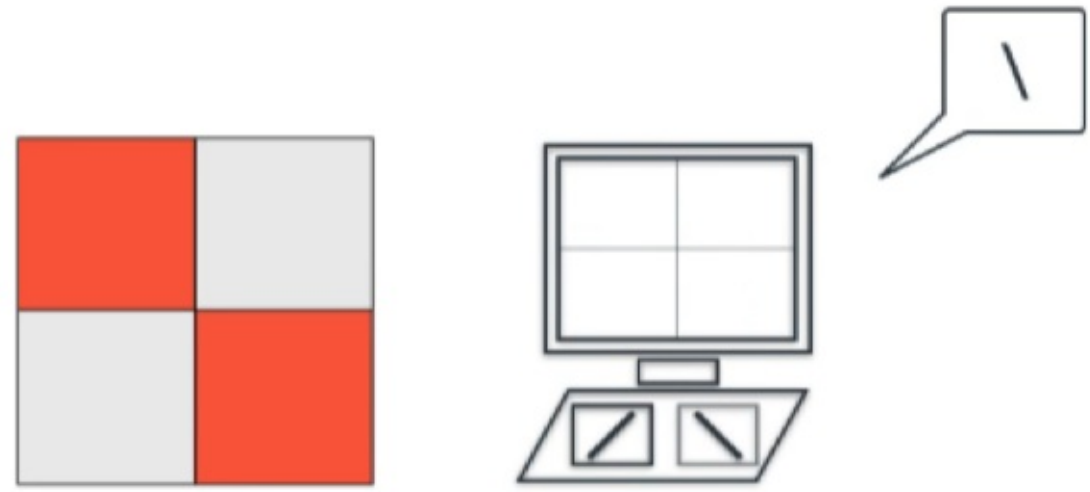
আমরা নিউরাল নেটওয়ার্ক দিয়ে তাহলে ডিজিটাল ফটো অ্যানালাইসিস করতে পারি। বলতে গেলে – ফটো রিকগনিশনের কাজ করতে পারি। আর নিউরাল নেটওয়ার্কের মাধ্যমে ইমেজ ক্লাসিফিকেশনের জন্য বহুল ব্যবহৃত একটা টেকনিক/মডেল হচ্ছে কনভলিউশনাল নিউরাল নেটওয়ার্ক। নিউরাল নেটওয়ার্ক কি সেটা আমরা ইতোমধ্যে জেনেছি। এবার আমরা জানবো কনভলিউশন মানে কি এবং এটার প্রয়োজনীয়তা। বলে রাখা ভালো এই কনভলিউশন টার্মটা কিন্তু শুধু নিউরাল নেটওয়ার্ক বা ইমেজ প্রসেসিং এর সাথে সম্পর্কিত কোন টার্ম না। বরং এটা একটা জেনেরিক ম্যাথেমেটিক্যাল টার্ম। সিগনাল প্রসেসিং –এও এর ব্যবহার দেখা যায় (বিশেষ টাইপের একরকম সিগনালও কিন্তু ঘুরে ফিরে ম্যাট্রিক্স ইমেজের মতই। যাহোক সময় থাকলে সেটা পরে উদ্ধার করবো □)।

সহজ উদাহরণ

চলুন আমরা একটা জগতের কথা চিন্তা করি যে জগতে একটা কম্পিউটার আছে যার ডিসপ্লে মাত্র 2x2 পিক্সেল সাইজের। আর ওই জগতে সব মিলে ২টা ক্যারেক্টার বা বর্ণ আছে যেগুলো হচ্ছে, \ এবং / অর্থাৎ ওখানকার কিবোর্ড দিয়ে একটা ফরওয়ার্ড স্ল্যাস অথবা একটা ব্যাক স্ল্যাস লেখা যায় :P অনেকটা নিচের মত -</p>

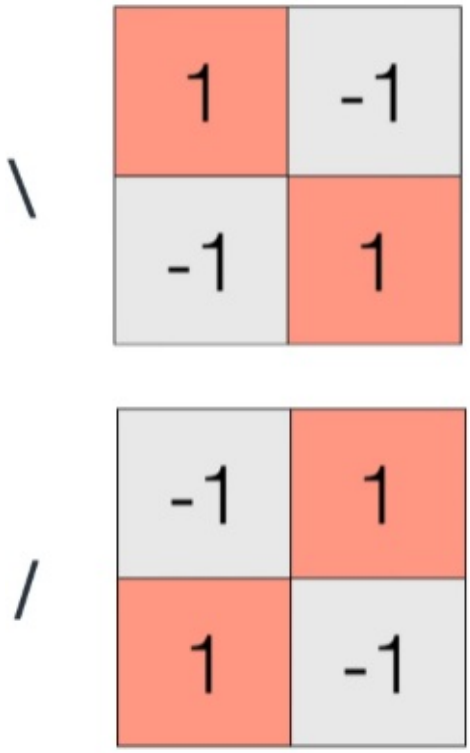


এখন আমরা এই কম্পিউটারের জন্য একটা ইমেজ ক্লাসিফায়ার সফটওয়্যার বানাতে চাই যাতে করে যখন আমরা কম্পিউটারকে নিচের ইমেজটি ইনপুট দেবো তখন সে যাতে সেটা চিনে বলতে পারে যে এটা একটা \. (ব্যাক স্ল্যাস)

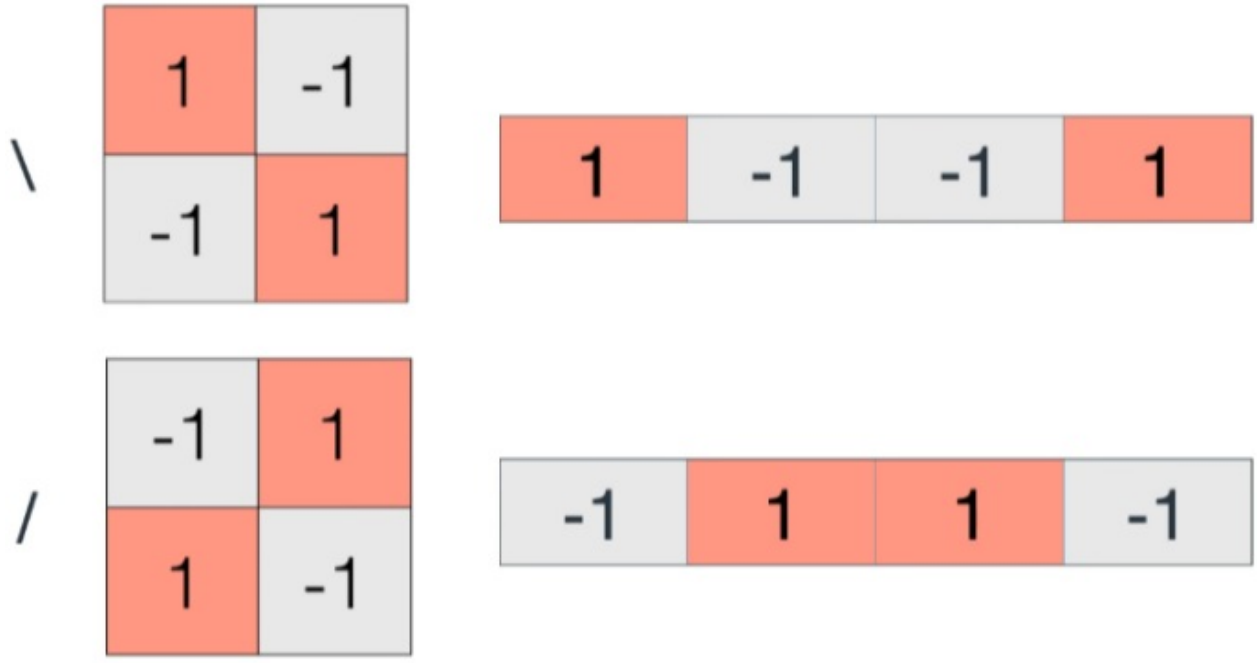


এবং একই ভাবে ফরওয়ার্ড স্ল্যাসটাকেও চিনতে পারে।

কিন্তু আমরা সবাই জানি যে কম্পিউটারের কাছে সব কিছুই আসলে নাম্বার। কম্পিউটার তো আর সরাসরি ফটো চেনে না, খুব বেশি হলে একটা ফটোর ম্যাথেমেটিক্যাল রিপ্রেজেন্টেশন চেনে। তাই, ধরে নিচ্ছি যে সেই কম্পিউটারের গ্রাফিক্স প্রোগ্রামটি মনিটরে কোন অ্যাক্টিভ পিক্সেল বা রং (এক্ষেত্রে ধরি লাল রঙের মান 1) দেখানোর জন্য 1 ব্যবহার করে এবং রং এর অনুপস্থিতি (ধরি গ্রে কালার এর মান -1) প্রকাশের জন্য -1 ব্যবহার করে।এতে করে প্রত্যেকটি বর্ণের গ্রাফিক্যাল রিপ্রেজেন্টেশন নিচের মত ধরে নেয়া যেতে পারে।



কিন্তু গ্রাফিক্যাল রিপ্রেজেন্টেশন বা ম্যাথেম্যাটিক্যাল রিপ্রেজেন্টেশন যাই হোক কম্পিউটার এই ম্যাট্রিক্সের মত দেখতে ডাটা ফরম্যাটকে লম্বা একটা অ্যারের মত করেই ধরে, হিসাব নিকাশ করার জন্য। অর্থাৎ নিচের মত,



পাইছি এবার। খেয়াল করুন এই দুটা আলাদা ইমেজের জন্য যে ফাইনাল ডাটা স্ট্রাকচার দাঁড়ালো সেগুলোকে কিন্তু আমরা আমাদের সেই প্রথম দিকের ইনপুট কম্বিনেশন সমস্যার সাথেই তুলনা করতে পারি। অর্থাৎ 1 -1 -1 1 কম্বিনেশনের আউটপুট ধরি 1 (ধরি 1 মানে ব্যাক স্ল্যাস) আর -1 1 1 -1 কম্বিনেশনের আউটপুট ধরি 0 (ধরি 0 মানে ফরওয়ার্ড স্ল্যাস)।

তাহলে যদি আমরা এই ইনপুট কম্বিনেশন গুলোতে সঠিক ভাবে কিছু ওয়েট সেট করতে পারি এবং দুইটা আলাদা ইনপুট কম্বিনেশনের জন্য দুটো আলাদা রেজাল্ট আসে (একটা শূন্যের কাছা কাছি আরেকটা এক এর কাছাকাছি। তাহলেই কিন্তু আমাদের ইমেজ ক্লাসিফায়ার তৈরি হয়ে গেলো। ওই ক্লাসিফায়ারে যতবারই [1 -1 -1 1] তথা ব্যাক স্ল্যাসের ইমেজের একটি ডাটা ম্যাট্রিক্স দেয়া হবে ততবার সে ওই ইনপুট গুলোর সাথে ওয়েট এর গুন যোগ করে আউটপুট দিবে ১ এর কাছাকাছি একটা মান আর আমরা জানি ১ মানে ব্যাক স্ল্যাস। তাই চিনে যাবো এই ইমেজ একটা ব্যাক স্ল্যাস। তো, চলুন এই সিম্পল ক্লাসিফায়ারের জন্য কিছু অপ্টিমাইজড ওয়েট বের করি।

প্রথমেই সব ওয়েট মনে করি 1. তাহলে [1 -1 -1 1] এর জন্য -

(1*1) + (-1*1) + (-1*1) + (1*1) = 1 - 1 - 1 + 1 = 0

আর [-1 1 1 -1] এর জন্য -

(-1*1)+(1*1)+(1*1)*(-1*1) = -1 + 1 + 1 - 1 = 0

লাভ হল না, দুইটার ক্ষেত্রেই শূন্য আসছে অর্থার ক্লাসিফাই করার মত সলিড Rule এটা হয় নি বা বলা যায় ওয়েট গুলো সঠিক হয় নি। ওয়েট গুলো একটু অ্যাডজাস্ট করে নিচের মত ট্রাই করি। অর্থাৎ প্রথম এইজের ওয়েট ধরি 1, দ্বিতীয় এইজের ওয়েট ধরি -1, তৃতীয় এইজের ওয়েট ধরি -1 এবং চতুর্থ এইজের ওয়েট ধরি 1. (যেহেতু চারটা Edge ধরে ৪টা ইনপুট আসছে একটা নিউরনের দিকে)

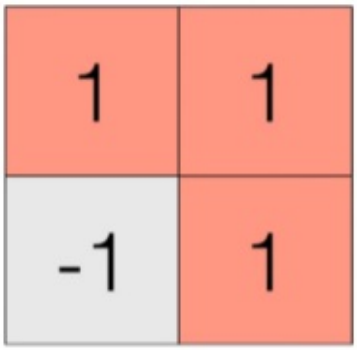
তাহলে [1 -1 -1 1] এর জন্য -

(1*1) + (-1*-1) + (-1*-1) + (1*1) = 1 + 1 + 1 + 1 = 4 Sigmoid করে 0.98

আর [-1 1 1 -1] এর জন্য -

(-1*1)+(1*-1)+(1*-1)*(-1*1) = -1 - 1 - 1 - 1 = -4 Sigmoid করে 0.01

অর্থাৎ আমরা মোটামুটি সঠিক ওয়েট এর খোঁজ পেয়েছি যেগুলো একটি ব্যাসিক নিউরাল নেটওয়ার্কে ব্যবহার করলে দুটা আলাদা ইনপুট কম্বিনেশনের জন্য দুটো আলাদা আউটপুট আসে এবং তার সাহায্যে আমরা দুটো ইমেজকে ক্লাসিফাই বা আলাদা করে চিনতে পারি :D একটু পরীক্ষা করেও দেখতে পারেন যে আসলেই সঠিক ওয়েটের খোঁজ পাওয়া গেছে কিনা। যেমন - নিচের দুটো ইমেজের কথা চিন্তা করুন,



দেখে মনে হচ্ছে উপরের ইমেজটি একটি ব্যাক স্ল্যাস কিন্তু উপড়ে একটু কালি বেশি পরে গেছে। আর নিচেরটা একটা ফরওয়ার্ড স্ল্যাস কিন্তু উপড়ে পুরোটা আকা শেষ হয়নি। এটা আমরা ধারনা করতে পারছি। কিন্তু চলুন পরীক্ষা করে দেখি আমাদের একটু আগের ব্যাসিক ইমেজ ক্লাসিফায়ার নিউরাল নেটওয়ার্ক এটা ধরতে পারে কিনা।

তাহলে প্রথম ইমেজ অর্থাৎ [1 1 -1 1] এর জন্য -

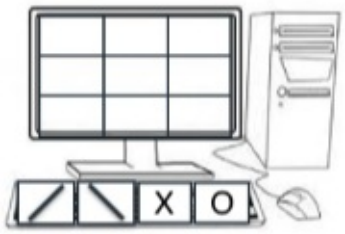
(1*1) + (1*-1) + (-1*-1) + (1*1) = 1 - 1 + 1 + 1 = 2 Sigmoid করে 0.88 (1 এর কাছাকাছি অর্থাৎ ব্যাক স্ল্যাস)

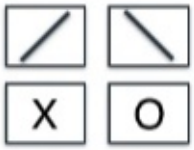
আর [-1 -1 1 -1] এর জন্য -

(-1*1)+(-1*-1)+(1*-1)*(-1*1) = -1 + 1 - 1 - 1 = -2 Sigmoid করে 0.11 (0 এর কাছাকাছি অর্থাৎ ফরওয়ার্ড ল্যাস)

গ্রেট, এরকম ভাঙ্গা হাতের লেখাকেও আমাদের ক্লাসিফায়ার ঠিকি ধরতে পারছে :) কিন্তু ভুলে গেলে চলবে না, এই যে, হুট করে সব ওয়েট [1 1 1 1] এর কথা চিন্তা করে তারপর অপ্টিমাইজড ওয়েট [1 -1 -1 1] -এ আসতে পারা, এখানে কিন্তু সেই এরর, অ্যাডজাস্ট, লার্নিং লুপেরই কাজ করতে হবে। এসব আগেই আলোচনা হয়ে গেছে :) আমরা বোঝার জন্য এক লাফেই অপ্টাইজ ওয়েট নিয়ে পরীক্ষা করেছি।

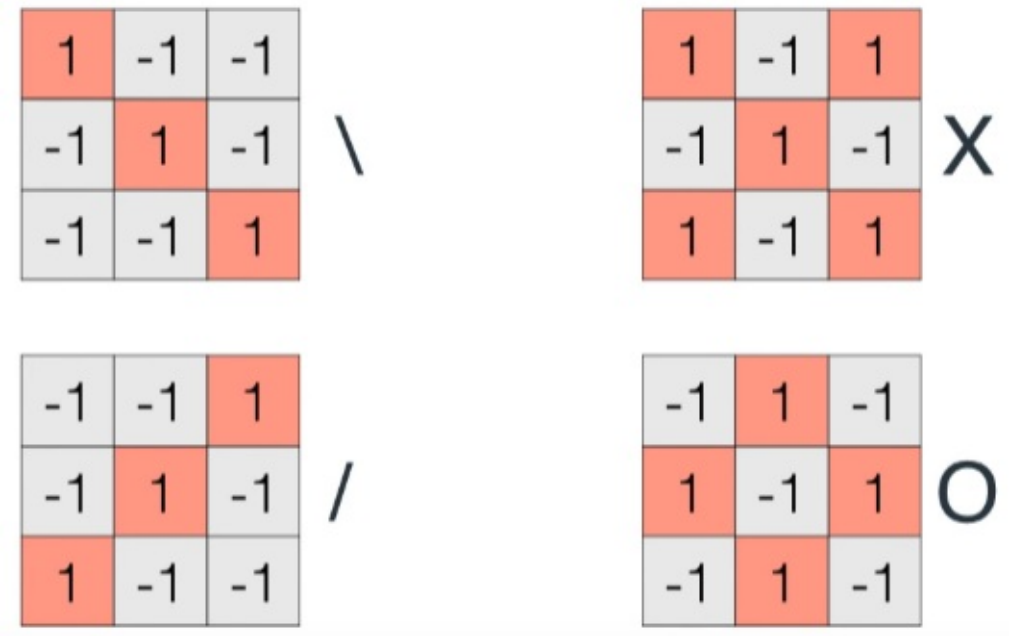
এবার চলুন একটু জটিল আরেকটা জগতের কথা চিন্তা করি যে জগতে একটা কম্পিউটার আছে যার ডিসপ্লে 3x3 পিক্সেল সাইজের। আর ওই জগতে সব মিলে ৪টা ক্যারেক্টার বা বর্ণ আছে যেগুলো হচ্ছে - \ / X O অর্থাৎ ওখানকার কিবোর্ড দিয়ে একটা ফরওয়ার্ড স্ল্যাস, একটা ব্যাক স্ল্যাস, একটা এক্স, আরেকটা ও লেখা যায় :P অনেকটা নিচের মত -





Computer Alphabet

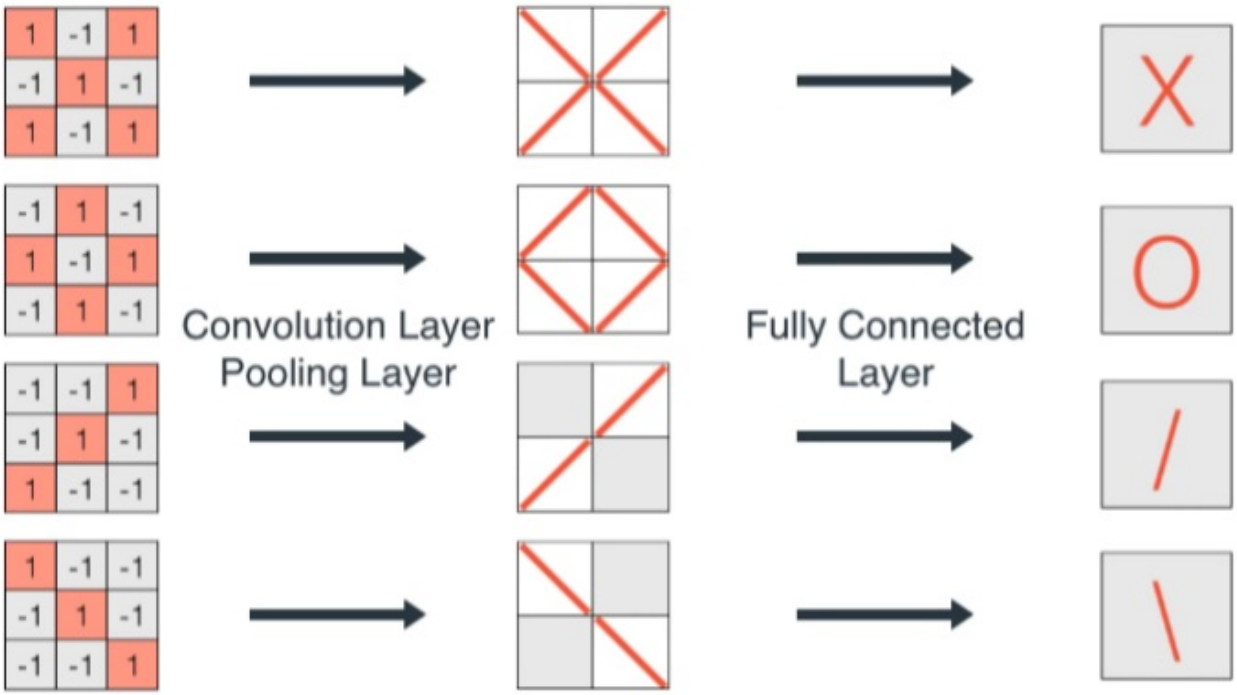
এখন মনে করুন যে ওই ৪টি বর্ণের কোন একটিকে কম্পিউটারে টাইপ করলে এর মনিটরে নিচের মত করে সেই ক্যারেক্টারটি রেন্ডার হয় -



এখন প্রশ্ন হচ্ছে কিভাবে আমরা তুলনামূলক জটিল একটি ক্লাসিফায়ার বানাতে পারি যার মাধ্যমে আমরা এই ক্যারেক্টার গুলোকে শুধুমাত্র তাদের ডাটা দেখে আলাদা করে চিনতে পারি? এখানে কিন্তু সেই ব্যাসিক ক্লাসিফায়ার কাজ করবে না। কেন করবে না সেটা এই পোস্টের দ্বিতীয় উদাহরণ পড়লেও আন্দাজ পাওয়া যাবে। এমনকি খুব সময় দিয়ে ওয়েট অ্যাডজাস্ট করে করেও এর জন্য পারফেক্ট কোন ওয়েট ম্যাট্রিক্স পাওয়া যাবে না যাতে করে ক্লাসিফায়ারটি খুব অ্যাকিউরেট হয়। কারন - বাস্তবে একটা বর্ণ একটা ইমেজের একদম মাঝেখানেই যে থাকবে তা নয়। বর্ণ একটাই যদি ধরি X. সেটা এক একটা ইমেজে এক এক ভাবে থাকতে পারে। যেমন - কোন ফটোতে একটু ডানে চাপিয়ে, কোনটাতে একটু বাপে চেপে অথবা কোথাও একটু ছোট (কম পিক্সেল ডাইমেনশন নিয়ে), অথবা কোথাও বড় আকারে থাকতে পারে। তাই পিক্সেল গুলোর সাথে যতই পারেফেক্ট ওয়েট জুড়ে দেয়া হোক না কেন, অবস্থা বিশেষে ক্লাসিফায়ার চিনতে পারবে না যে সেই ফটোতে নির্দিষ্ট একটা বর্ণ আছে।

তো উপায় যেটা সেটা হচ্ছে, একটা X দেখতে কেমন এবং তার বিভিন্ন অংশ গুলো কেমন সেগুলোকে প্রথমে চিনে তারপর অংশ ভিত্তিক কম্বিনেশন খুঁজতে হবে সেই বড় ইমেজটাতে (যেটায় X লুকায় আছে)। পিক্সেল ভিত্তিক খোঁজে আর কাজ হবে না। এতে করে X বর্ণটা ইমেজের যেখানেই থাকুক আর যেভাবেই থাকুক না কেন, তার অংশ বিশেষগুলো কিন্তু যেখানে থাকবে একভাবে থাকবে। অর্থাৎ X এর হাত গুলো কিন্তু ছিন্ন বিচ্ছিন্ন হয়ে থাকবে না। যেখানেই থাকুক না কেন হাত গুলো জয়েন্ট আকারেই থাকবে। তেমনি O যেখানেই যেভাবে থাকুক, মাঝখানে একটা ফাকা জায়গা থাকবে এটা স্বাভাবিক।

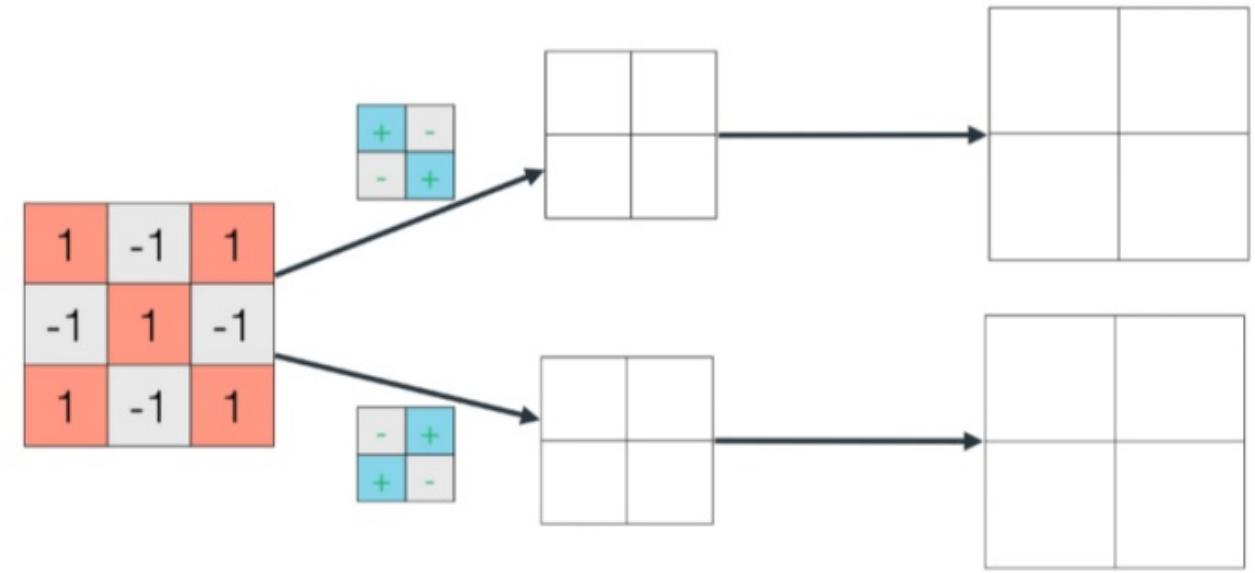
আচ্ছা, এবার আমরা একটু 2x2 পিক্সেলের জগত থেকে হেল্প নেবো। সেখানে আমাদের ব্যাক এবং ফরওয়ার্ড স্ল্যাস আছে। খেয়াল করুন, সেগুলাই কিন্তু X এর হাত গুলোর মতই। অর্থাৎ একভাবে খেয়াল করলে বলা যায় যে - তিন পিক্সেল ওয়ালা জগতের X, O, \, / গুলো বস্তুত দুই পিক্সেল ওয়ালা জগতের \ এবং / এর সমন্বয়েই তৈরি। নিচের ফিগারটা একটু খেয়াল করি,



উপরের ডায়ারগ্রামটাই কিন্তু একটা কনভলিউশনাল নিউরাল নেটওয়ার্কের ব্লক ডায়াগ্রাম। (ভয়ের কিছু নাই বিস্তারিত এড়িয়ে যাওয়া হবে না)। দুটি স্টেজে একটা কনভলিউশনাল নিউরাল নেটওয়ার্কের কাজকে বর্ণনা করা যায়। প্রথম স্টেজে অর্থাৎ বামের দিকে একটি বড় ইমেজ থেকে Convolution এবং Pooling টেকনিকের মাধ্যমে ছোট কিন্তু অর্থবহ ইমেজ বের করে আনা হয়। Convolution এর জন্য ছোট ইমেজ ব্যবহার করা হয় যেটাকে ফিল্টার বলা হয়। যেমন X ওয়ালা বড় একটা ইমেজের উপর আমরা \ এর ইমেজ এবং / এর ইমেজ নিয়ে Convolution করতে পারি কারণ পুরো X এর মধ্যে বস্তুত এগুলোরই অবস্থান। আর তাই X এর হাত গুলোকে চিনে নেয়ার জন্য আমরা শুধু (কাটা) হাতের ফটোকে ফিল্টার হিসেবে ব্যবহার করতে পারি।

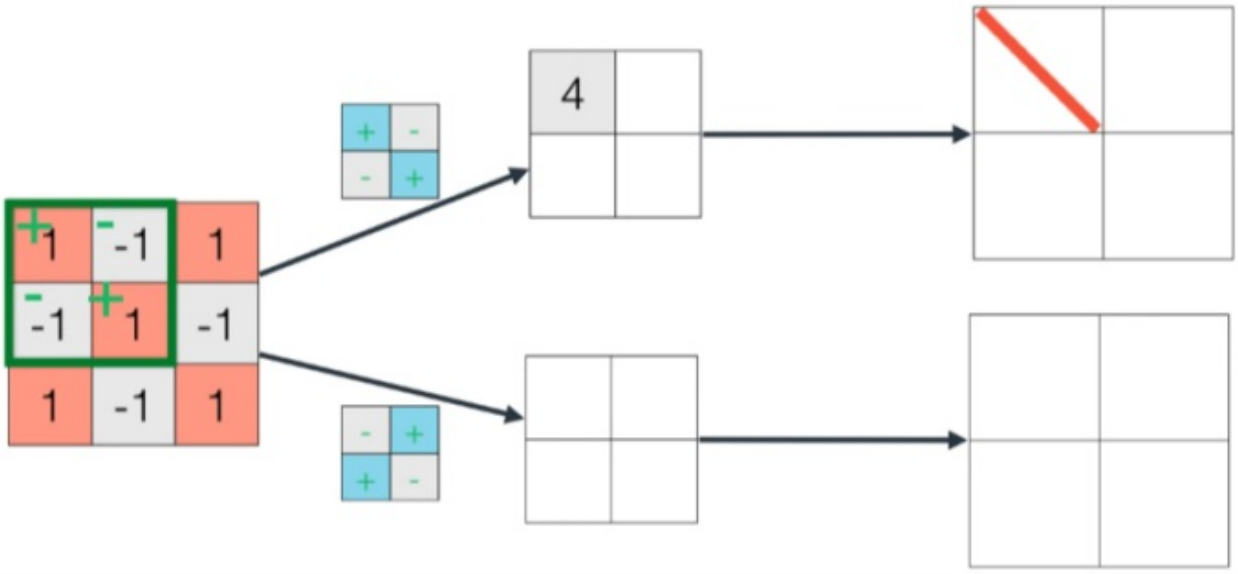
এরপর সেই ছোট আকারের ইমেজ গুলোকে নিয়ে পরের স্টেজে নিউরাল নেটওয়ার্ক এর কাজ করা হয়। এই স্টেজে নিউরাল নেটওয়ার্কের লজিক, ওয়েট ইত্যাদি ফিক্স করে, এর আগের স্টেজের ইমেজ অর্থাৎ Pooled ইমেজের উপর কাজ করা হয় এবং পিক্সেল কম্বিনেশন যাচাই করে পার্থক্য বা চেনার কাজ করা হয়। এই দ্বিতীয় স্টেজটাকে বলে Fully Connected Layer আর প্রথম স্টেজকে বলে Convolution Layer (এই স্টেজে পুলিং বা অন্যান্য আরও কিছু ফাংশনের ব্যবহার থাকতে পারে)।

একটু বিস্তারিতভাবে এবার এই কাজ গুলো করে দেখি। নিচের ফিগারটি খেয়াল করুন,

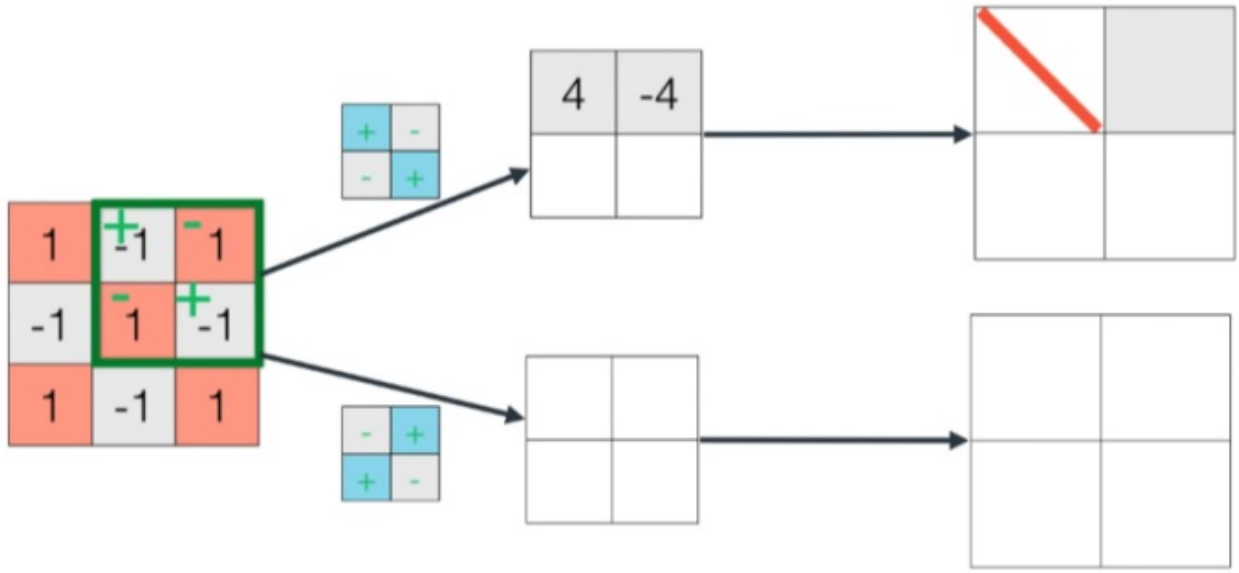


এখানে বাম পাশে একটা X ওয়ালা ইমেজ আছে। তার ডানে আমরা দুটো ফিল্টার নিয়েছি। একটা ফিল্টার আসলে ব্যাক স্ল্যাস চেনার ফিল্টার। এই ফিল্টার দিয়ে বড় ইমেজের উপর ঘুরে বেড়ালে (Convolution করলে) যেখানেই হাই ভ্যালু পাবে সেখানেই একটা ব্যাক স্ল্যাস পাওয়া গেছে বলে ধরে নেয়া যাবে। আগের মত চিন্তা করলে এটা ব্যাক স্ল্যাস চেনার সেই ওয়েট ম্যাট্রিক্স। শুধু +1 আর -1 কে + এবং - এ চিহ্নিত করা হয়েছে (অর্থাৎ [1 -1 -1 1] => [+ - - +]. আরেকটা ফিল্টার হচ্ছে ফরওয়ার্ড স্ল্যাস চেনার ফিল্টার। এই ফিল্টারকে বড় ইমেজের উপড়ে ঘুরিয়ে নিয়ে বেড়ালে যখনই হাই ভ্যালু পাবে তখন সেখানে একটা ফরওয়ার্ড স্ল্যাস আছে বলে জানা যাবে।

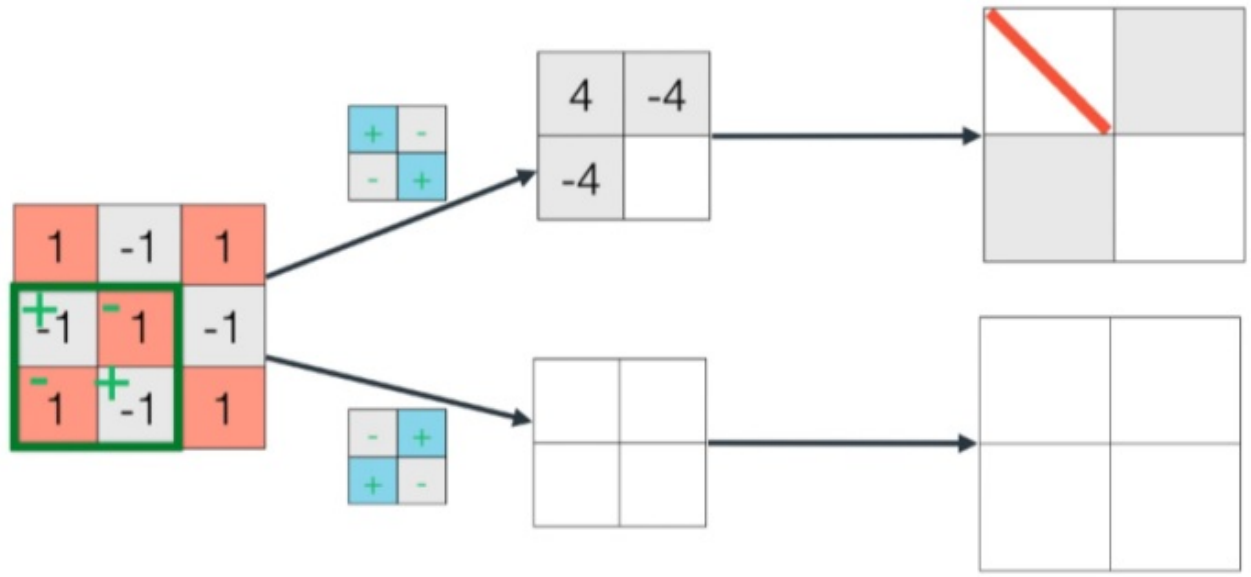
Convolution শুরু করা যাক



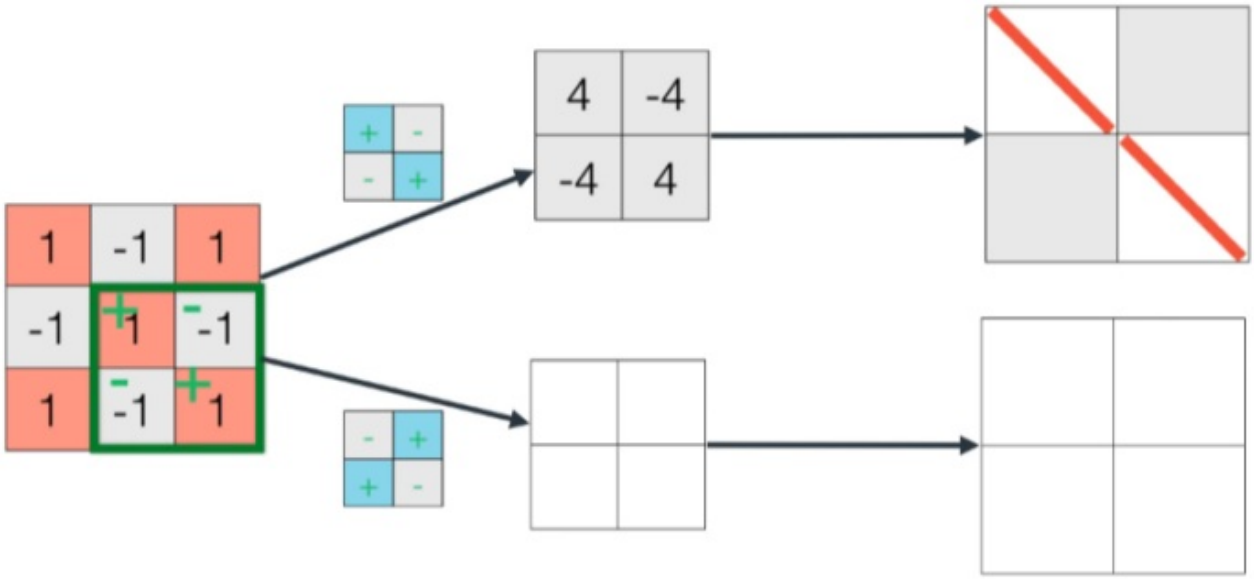
প্রথমেই আমরা উপড়ের ফিল্টার কে সাথে নিয়ে বড় ইমেজের টপ লেফট সাইডে বসিয়েছি এবং ওখানকার পিক্সেল ভ্যালুর সাথে ফিল্টার এর সমন্বয় ঘটিয়ে (গুন যোগ) ভ্যালু পেয়েছি 4. পরের ধাপে বলা হচ্ছে যদি তুমি একটা হাই ভ্যালু পাও তার মানে হচ্ছে তুমি যে অংশ (অঙ্গ) খুজতেছিলা সেটা পাওয়া গেছে। অর্থাৎ Convolution এর এই ধাপে একটা \ পাওয়া গেছে। কারণ \ এর ফিল্টার দিয়ে চেক করা হচ্ছিল এবং ভ্যালু হাই এসেছে। এরপর ফিল্টারকে টপ রাইটে স্লাইড করে নিয়ে যেতে হবে নিচের মত,



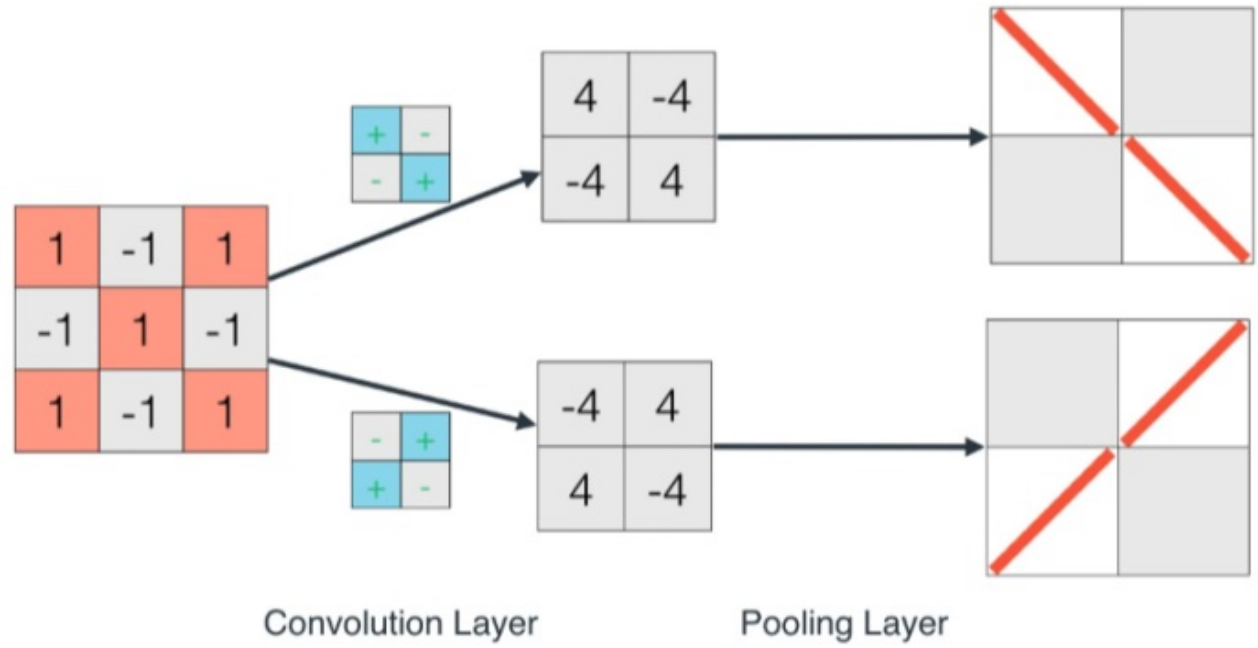
এই অবস্থায় ভ্যালু এসেছে -4 যা আমাদের ফিল্টার দিয়ে অংশ চেনার লজিক অনুযায়ী মিথ্যা। তাই পরের ধাপ বলছে আমি কোন ব্যাক স্লাসের অস্তিত্ব পাই নাই তাই Pooled ইমেজে গ্রে বসায় রাখলাম। আবার ফিল্টারকে স্লাইড করে বোটম লেফট কর্নারে সেট করলাম এবং নিচের অবস্থা পেলাম,



আবারও ফিল্টারকে বোটম রাইট কর্নারে নিয়ে সেট করলাম এবং নিচের মত অবস্থা পর্যবেক্ষণ করলাম যে অনুযায়ী বলা যায় - এইবার আবারও একটা \ এর অস্তিত্ব পাওয়া গেছে।



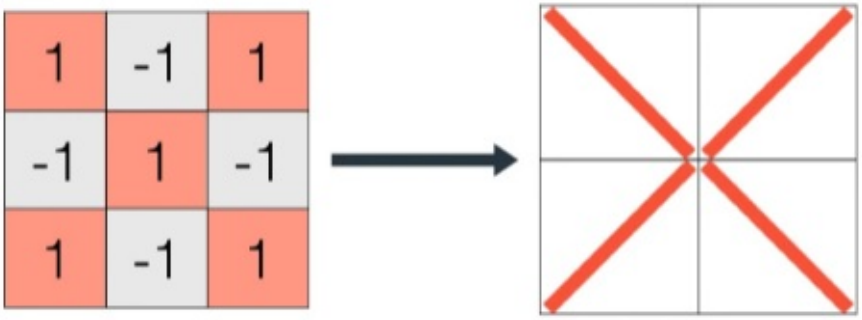
এবার একই ভাবে আমরা ফরওয়ার্ড স্ল্যাস খোজার ফিল্টার দিয়ে বড় ইমেজের উপর Convolution করে নিচের মত আপডেট পাবো। অর্থাৎ যেখানে যেখানে ফিল্টার এর সমন্বয় হাই ভ্যালু পেয়েছে সেখানে একটি করে / এর অস্তিত্ব পাওয়া গেছে,



মনে রাখবেন, প্রত্যেকটি ফিল্টারই কিন্তু উপরে বাম থেকে শুরু করে, উপড়ে ডানে এবং তারপর নিচে বাম থেকে, নিচে ডানে গিয়ে কাজ/চেক/কনভলিউশন শেষ করে।

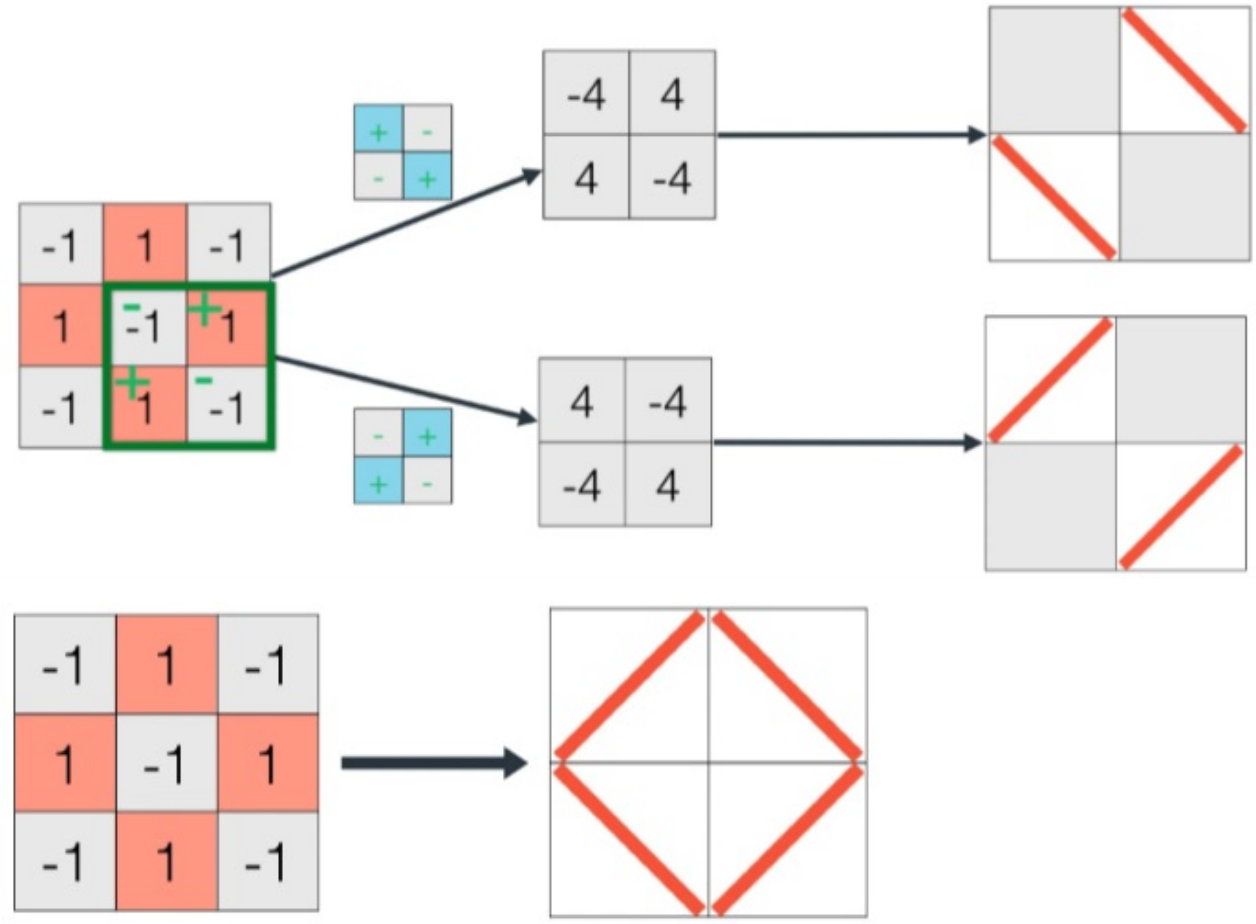
আর হ্যা, এই যে ফিল্টারকে স্লাইড করে পিক্সেল-কম্বিনেশন/অংশ/অঙ্গ খোজার স্টেজ, সেটাই কনভলিউশনাল লেয়ার। তারপর, আপনার ডিফাইন করা একটা নির্দিষ্ট থ্রেসহোল্ড ভ্যালুর চেয়ে বড় ভ্যালু আসলে (যেমন এক্ষেত্রে ধরছি 3) যে আপনি ভেবে নিচ্ছেন 'একটা অংশ আপনি খুঁজে পেয়েছেন' আর ভ্যালু কম আসলে ধরে নিচ্ছে 'ওখানে সেই অংশ পাওয়া যায় নি' এটাই হচ্ছে পুলিং লেয়ারের কাজ।

এরপর ডান পাশের অর্থাৎ পুলিং লেয়ার থেকে পাওয়া দুটো আলাদা ইমেজকে আমরা একসাথে করতে পারি নিচের মত,



অর্থাৎ এখন থেকে কম্পিউটার ভেবে নিবে যে বাম পাশের 3x3 পিক্সেলের যে ফটো সেটারই একটা সিমপ্লিফায়েড ভার্সন হচ্ছে ডান পাশের 2x2 পিক্সেলের ইমেজ যেটা কিনা কিছু \ / (সেই অংশ বা অঙ্গ যাই বলেন) এরসমন্বয়।

একই ভাবে আমরা O ওয়ালা বড় ইমেজকে ওই দুটো ফিল্টার দিয়েই Convolute করেও নিচের মত পুলিং লেয়ারের সাহায্যে সিমপ্লিফায়েড ভার্সনে কনভার্ট করতে পারি।

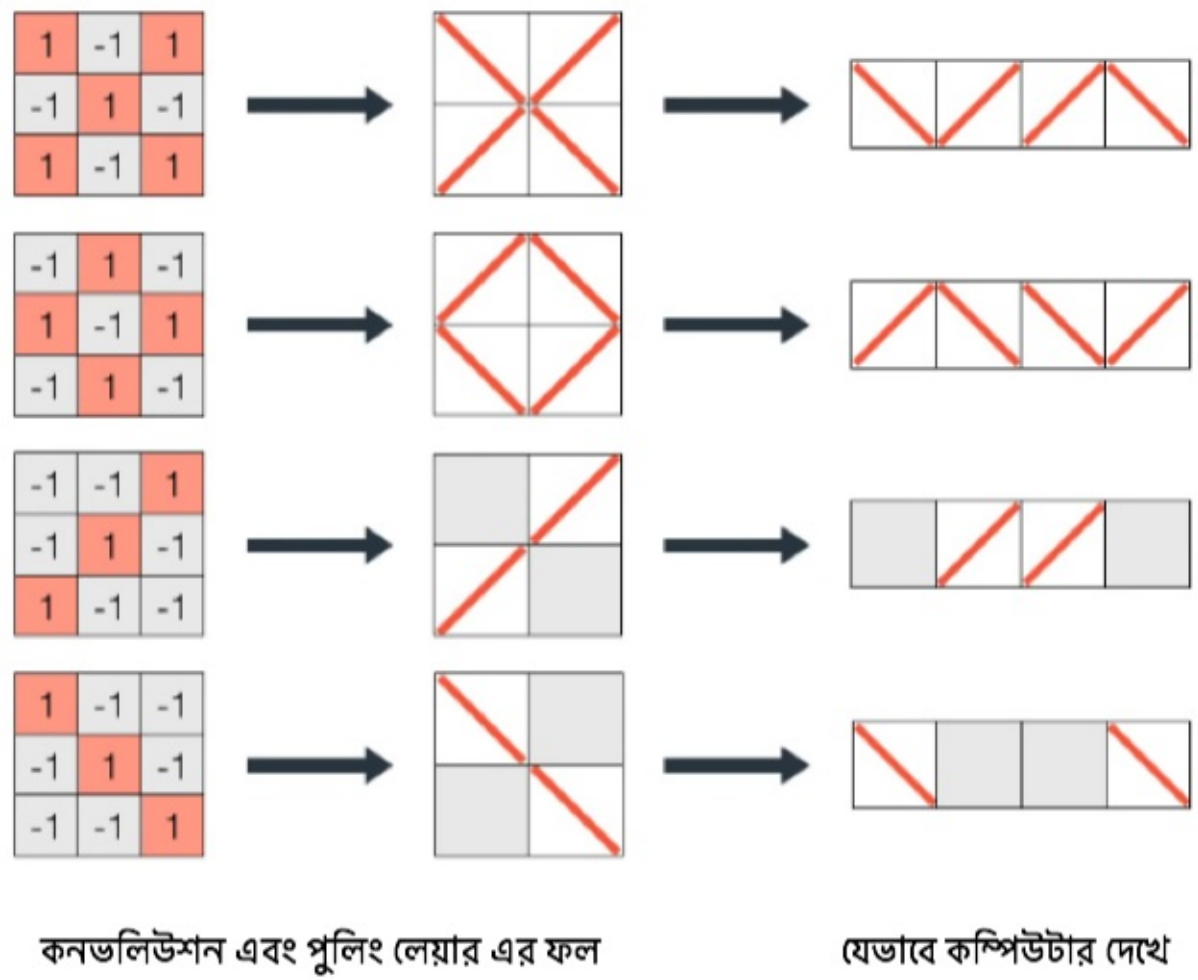


এভাবে আমাদের ৪ বর্ণ ওয়ালা জগতের বাকি দুটো বর্ণ \ এবং / এর জন্যও কাজ করে নিতে পারেন। সেক্ষেত্রেও তিন পিক্সেল ওয়ালা স্ল্যাসের সাপেক্ষে থ্রেসহোল্ড মেনে পুলিং লেয়ারের কাজ শেষে দুই পিক্সেল ওয়ালা স্ল্যাস পাবেন।

আবারও বলে নিচ্ছি, কনভলিউশন লেয়ারের কাজ হচ্ছে একটি ফিল্টার (র‍্যান্ডম বা নির্দিষ্ট) এর সাহায্যে একটি পূর্ণ ইমেজের উপর স্লাইড করে ঘুরে বেরিয়ে খুঁজে দেখা সেখানে কোথায় কোথায় ফিল্টার মোতাবেক অংশের অস্তিত্ব পাওয়া যায়। আর অস্তিত্ব আছে/নাই এর সিধান্ত নির্ভর করবে পুলিং লেয়ার এবং তার কাছে থাকা একটা থ্রেসহোল্ড ভ্যালুর উপর।

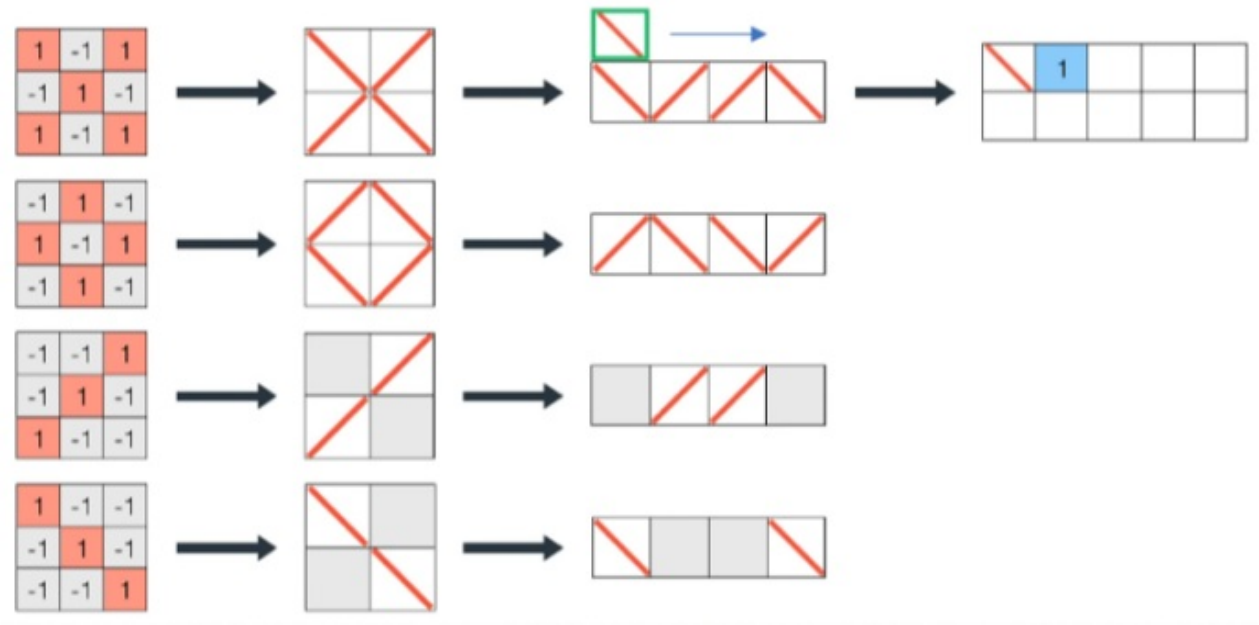
ফুলি কানেক্টেড লেয়ার

এবার আসুন এই লেয়ারের কাজ নিয়ে ভাবি - এই লেয়ার ইনপুট হিসেবে নেয় কনভলিউশন এবং পুলিং লেয়ার কাছ থেকে পাওয়া প্রসেসড ইমেজ এবং তার উপর চিন্তা (সেই চিন্তা) এবং ট্রেনিং করে প্যাটার্ন বুঝে নেয় এবং ঠিক করে কোনটা কোন ইমেজ।</p>

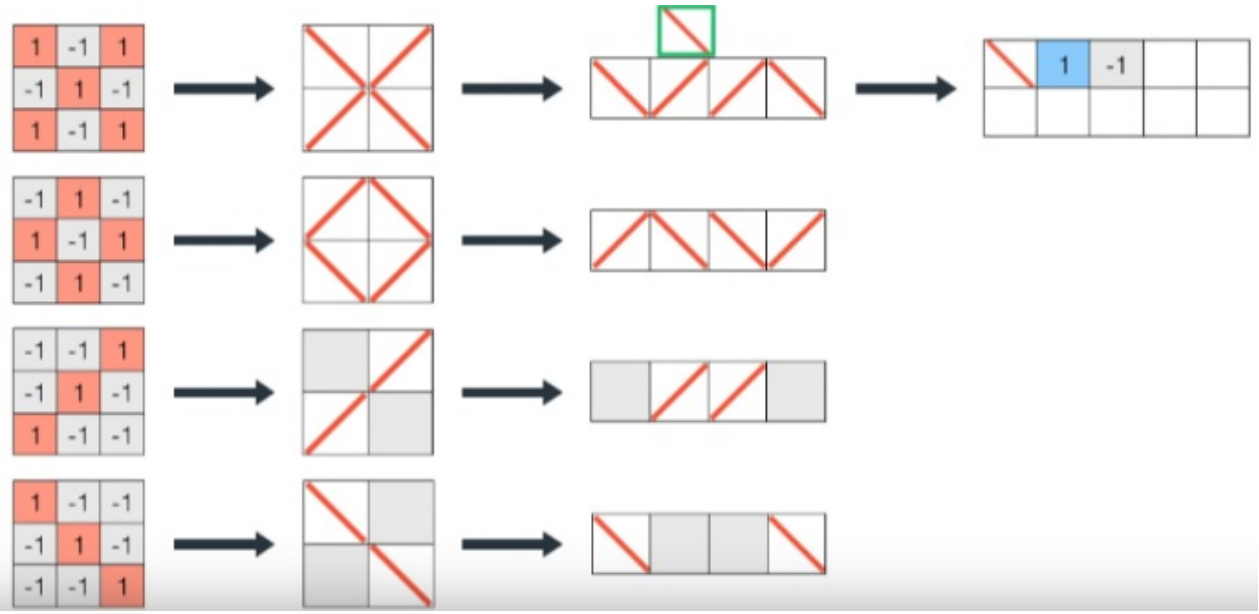


কনভলিউশন এবং পুলিং লেয়ার এর ফল যেভাবে কম্পিউটার দেখে

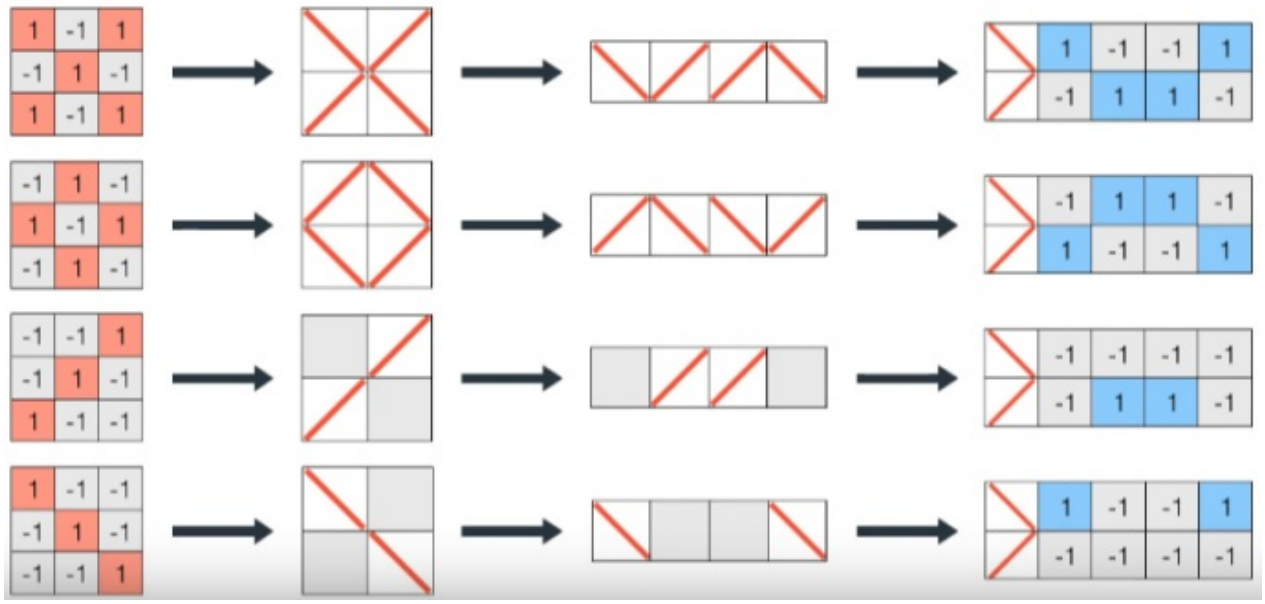
উপরে আবার আমরা একবার রিভিউ দিচ্ছি এখন পর্যন্ত কি কি হয়েছে। এই পর্যায়ে এসে আমরা আবারো এক ধরনের ফিল্টার তৈরি করবো। ফিল্টারটা এবার অনুমান করা না বরং উপরের স্টেজের ফাইনাল ধাপের যে ফ্ল্যাট রিপ্রেজেন্টেশন পাওয়া গেছে তার একটা অর্থবহ ম্যাট্রিক্স রূপ। অর্থাৎ, এই ম্যাট্রিক্সে আমরা সেই তথ্য স্টোরে করবো যার মাধ্যমে বলা যাবে কোথায় আমরা একটি ব্যাক স্ল্যাস এবং কোথায় একটি ফরওয়ার্ড স্ল্যাস পেয়েছিলাম। অর্থাৎ একটা রেকর্ড বা লগ রাখার মত। নিচের ফিগারে আমরা দেখবো কিভাবে এটা করতে পারি,



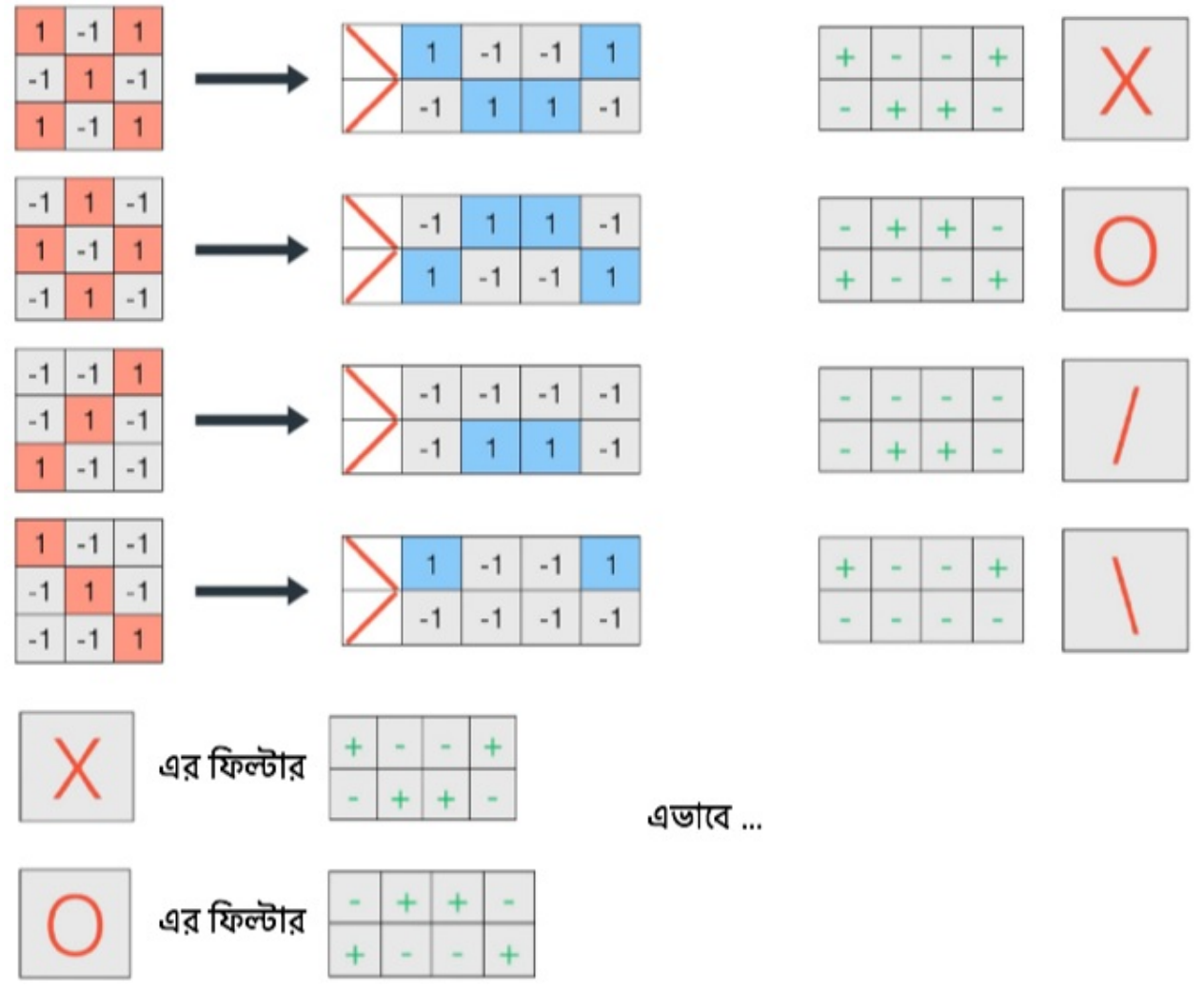
অর্থাৎ X এর পুলিং লেয়ারের পর ফ্ল্যাট রিপ্রেজেন্টেশনে প্রথমটা ব্যাক স্ল্যাস তাই আমরা স্টোর ম্যাট্রিক্সের প্রথম ভ্যালু সেট করলাম 1. আবার ফ্ল্যাট রিপ্রেজেন্টেশনের দ্বিতীয় এলিমেন্টটা ব্যাক স্ল্যাস না তাই স্টোরে ম্যাট্রিক্সের প্রথম সারির দ্বিতীয় কলামে -1 এভাবে,



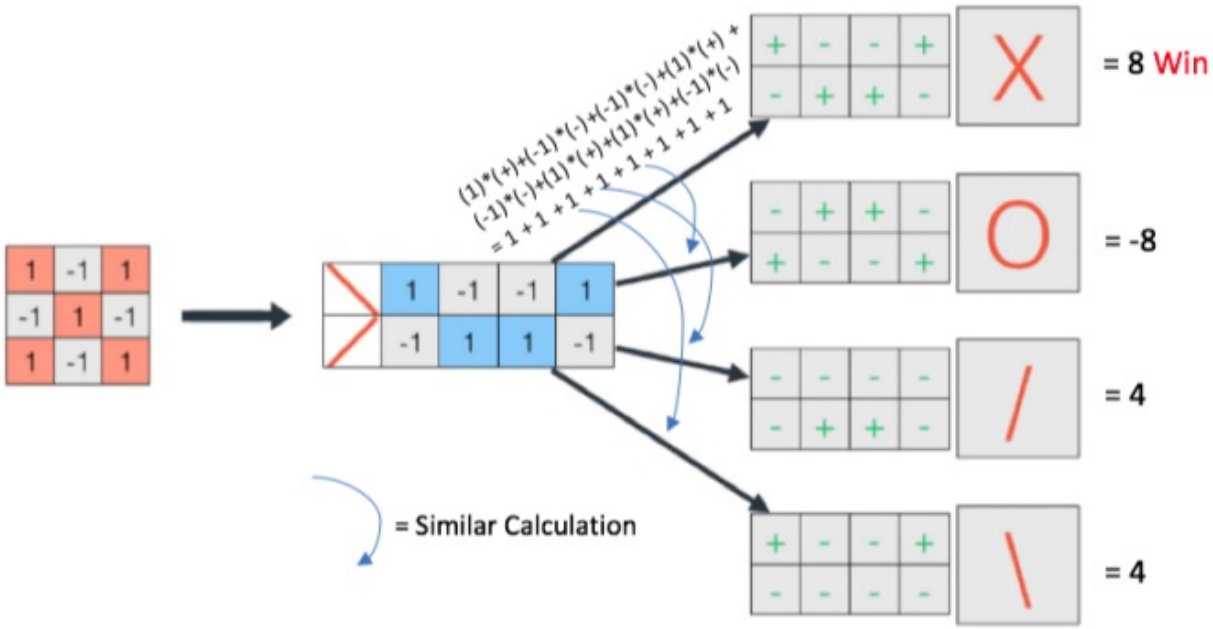
ফাইনালি আমরা সব গুলো ফিল্টার পাবো নিচের মত,



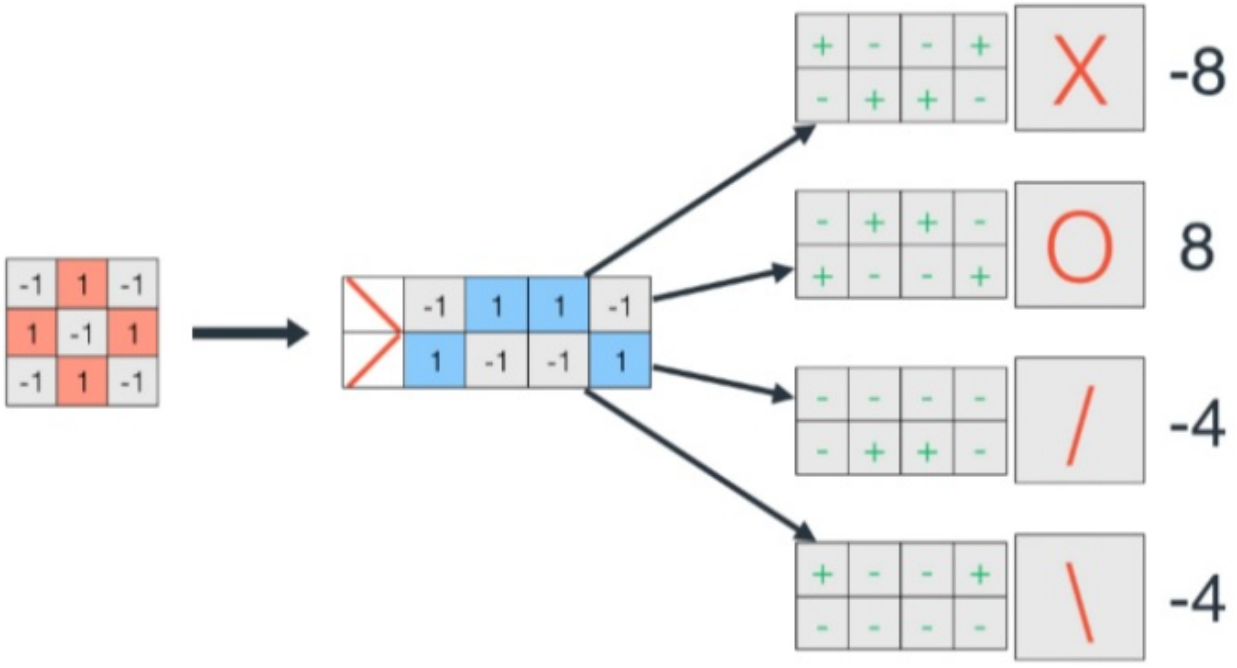
এখন আমরা ফিল্টারের 1, -1 কে যদি + এবং মাইনাসে প্রকাশ করি তাহলে ব্যবহার উপযোগী ফিল্টার ধরতে পারি নিচের মত,



এরপর আমরা যেটা করতে পারি তা হল, আমরা একটা নির্দিষ্ট বর্ণের জন্য প্রাপ্ত 2D ম্যাট্রিক্স এর সাথে সবগুলো ফিল্টার (+ - ওয়ালা ম্যাট্রিক্স) এর তুলনা করবো। যেমন - নিচে X এর জন্য প্রাপ্ত ম্যাট্রিক্স এর সাথে সবগুলা ফিল্টার এর সমন্বয় (গুন যোগ) করা হয়েছে



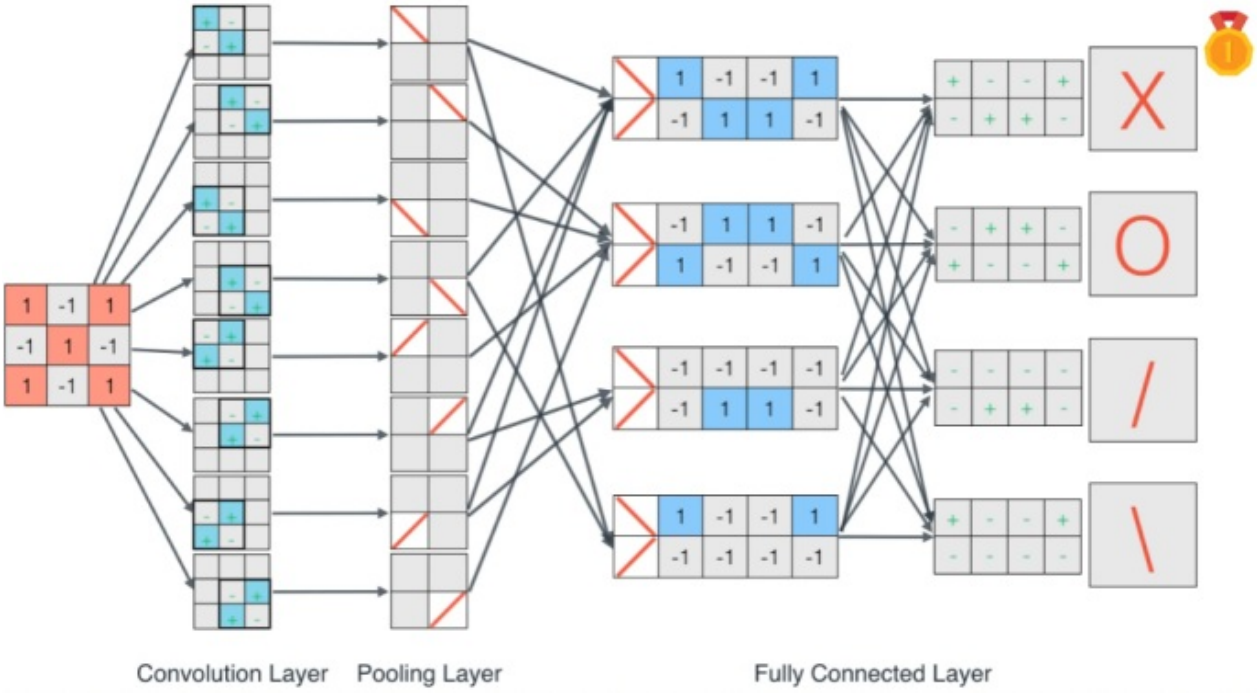
এবং X এর ফিল্টার এর সাথেই সব চেয়ে বেশি স্কোর এসেছে (আসাটাই স্বাভাবিক কারন 2D ম্যাট্রিক্স আর ফিল্টার ম্যাট্রিক্স একই দিকের মান নির্দেশ করে)। তাই বলা যায় এই ইমেজটি X এর ইমেজ :) :D আবার O এর জন্য এই ফুলি কানেক্টেড লেয়ারটির ক্যালকুলেশন ট্রাই করে দেখি,



এক্ষেত্রেও O এর ফিল্টারের সাথেই বেশি স্কোর আসছে তাই সঠিক উত্তর, O.

কমপ্লিট কনভলিউশনাল নিউরাল নেটওয়ার্ক

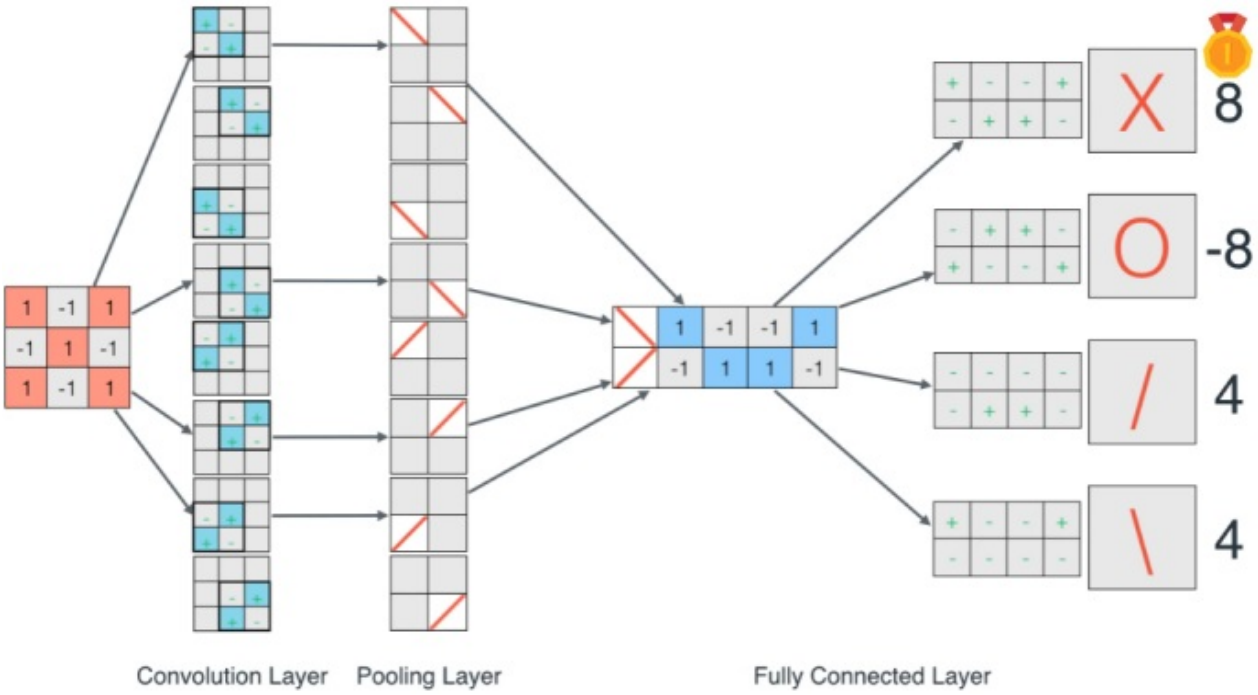
নিচের ফিগারটার দিকে একবার চোখ বুলাই -</p>



ডায়াগ্রামের একদম বামে আমাদের একটি ইনপুট ইমেজ। তার উপর আমরা দুটো ফিল্টার/মাস্ক অ্যাপ্লাই করেছি প্রত্যেকটা ৪ বার করে মোট ৮ বার (উপর নিচে ডানে বামে)। এটাই আমাদের কনভলিউশনাল লেয়ার। এরপর পুলিং লেয়ারে আমরা চিহ্নিত করেছি ক্ষুদ্র কিছু অংশ বিশেষ যার মাধ্যমে আমাদের আসল বর্ণ গুলো গঠিত এবং সেগুলোর অস্তিত্ব রেকর্ড করেছি। পুলিং লেয়ারের যে আউটপুট ইমেজ অর্থাৎ 2x2 সাইজের সিম্পল ইমেজ সেগুলোকে পিক্সেলেটেড থেকে অর্থবহ ম্যাট্রিক্সে কনভার্ট করা হয়েছে। এখান থেকেই ফুলি কানেক্টেড লেয়ার শুরু বা চিরচেনা নিউরাল নেটওয়ার্ক স্টাইলে ক্যালকুলেশন শুরু। আর তাই এই লেয়ারে, আগের ধাপে পাওয়া ম্যাট্রিক্স গুলোকে (প্রত্যেকটি) ডান পাশের প্রত্যেকটি ফিল্টার ম্যাট্রিক্স এর সাথে মাল্টিপ্লাই করা হয়েছে এবং আউটপুট লেয়ার তথা শেষ ধাপে স্কোর চেক করা হচ্ছে যে, কোন ফিল্টারের সাপেক্ষে স্কোর হাই।

পরীক্ষা করে দেখা যাক

আবার X এর ফটো ইনপুট হিসেবে দিয়েই পরীক্ষা করি পুরো কনভলিউশনাল নিউরাল নেটওয়ার্ক আসলেই X কে চিনতে পারে কিনা ।</p>

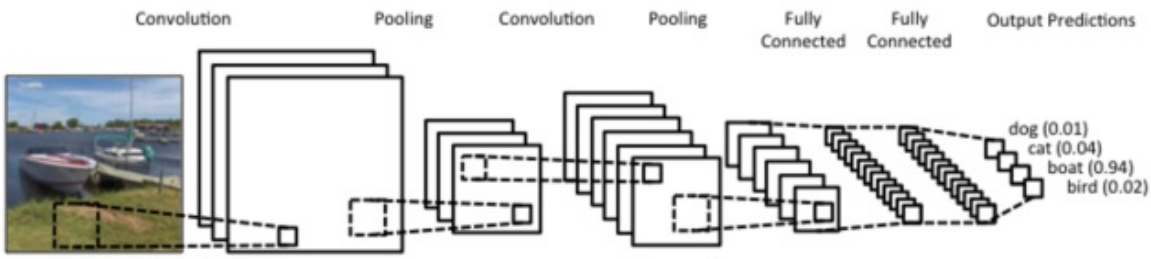


খেয়াল করুন, প্রথমে X এর ইমেজের উপর আমাদের \ / ফিল্টার চালিয়ে দেখা হয়েছে এবং পুলিং লেয়ারের কাছে শুধু ৪টি অবস্থা ভ্যালিড বা ফায়ার হয়েছে। ব্যাক স্ল্যাস ফিল্টার দিয়ে ঘোরার সময় যে দুটা অবস্থায় ব্যাক স্ল্যাস পাওয়া গেছে এবং ফরওয়ার্ড স্ল্যাস ফিল্টার দিয়ে ট্রাভেল করার সময় যে দুটা জায়গায় ফরওয়ার্ড স্ল্যাসের অস্তিত্ব পাওয়া গেছে (তাই পুলিং লেয়ারের ইনপুটে ৪টি কেইস)। এরপর পুলিং লেয়ার 2x2 পিক্সেলেটেড ইমেজ কে বিশেষ ম্যাট্রিক্সে কনভার্ট করছে এবং পরীক্ষার কোন এক সিচুয়েশনে এই ম্যাট্রিক্সটি ডান পাশের ফুলি কানেক্টেড লেয়ারের প্রত্যেকটি ফিল্টারের সাথে মাল্টিপ্লাইড হচ্ছে। ফাইনালি সে ডান পাশে অর্থাৎ আউপুটপুট লেয়ারে রেজাল্ট হিসেবে জানাচ্ছে তার পাওয়া স্কোর গুলো। আর স্কোর দেখে খুব সহজেই বুঝে নেয়া যাচ্ছে X এর উপরেই এই CNN এর কনফিডেন্স বেশি :)

বার বার মনে করিয়ে দিচ্ছি, এখানে বেশ কিছু হেল্পার ফাংশনের কাজ এড়িয়ে যাওয়া হয়েছে শুধু নিউরাল নেটওয়ার্কের ওয়ার্কিং প্রিন্সিপল সহজে বোঝানোর জন্য। যেমন - কম্পিউটারকে পারফেক্ট ফিল্টার বুঝতে, কনভলিউশন করাতে, ফুলি কানেক্টেড লেয়ারের ওয়েট/এইজ উদ্ধার করতে লক্ষ্য লক্ষ্য বার ঘুরে ফিরে কাজ করতে হয়। কারণ, শুরুতেই কম্পিউটার সব কিছুর জন্য (ফিল্টার, কনভলিউশন ইত্যাদি) র‍্যান্ডম কিছু ভ্যালু ধরে নেয়। তারপর ট্রেনিং ডাটা সেট এ যেহেতু প্রশ্ন উত্তর দুটাই আছে, তাই সেখান থেকে এরর কল্সিডার করে করে এবং সেই অনুযায়ী সব ভ্যালু অ্যাডজাস্ট করে করে ফাইনালি এরকম স্ট্যাবল একটা স্টেজে আসে। এরর এর উপর ভিত্তি করে ভ্যালু অ্যাডজাস্ট করা নির্ভর করে Gradient Descent এরউপর। এ সম্পর্কে বাংলায় পড়তে চাইলে এখানে ক্লিক করুন।

বাস্তব জগতে **CNN** অনেক তো খেলনা জগতের সমস্যা উদ্ধার করলাম আমরা। আসলেই রিয়েল লাইফ সিচুয়েশনে কিভাবে CNN কাজ করে তার একটা ধারনা নেই এখন। কারন, বাস্তবে কম্পিউটারে লক্ষ্য লক্ষ্য পিক্সেল যেমন আছে তেমনি সব ফটো আমাদের আরাম দেয়ার জন্য 3x3 পিক্সেল নিয়ে বসে নাই। কয়েক মেগা পিক্সেলের ইমেজ এখন

সবার কাছেই। তাই আমাদের যেমন নিউরাল নেটওয়ার্ক ডিজাইন করতে হবে খুব বুদ্ধি করে, তেমনি কম্পিউটারকেও রেডি থাকতে হবে বিনা ইস্যুতে কোটি কোটি বার একই বোরিং হিশাব করতে আর লুপের উপর থাকতে।



উপরে একটা পূর্ণ CNN এর ব্লক ডায়াগ্রাম দেখানো হয়েছে। প্রথমেই বাম পাশে একটি নৌকার ছবি ইনপুট দেয়া হচ্ছে এবং এই নেটওয়ার্কে দুই স্টেজে Convolution এবং Pooling এর কাজ করা হয়েছে (প্রয়োজনে আরও হতে পারে)। তো, প্রথম কনভলিউশন এবং পুলিং এর সময় এই ফটো থেকে কিছু গুরুত্বপূর্ণ পিক্সেল গুচ্ছ বা অবজেক্টের অংশ বিশেষ আলাদা করে নিয়ে নেয়া সম্ভব হয়। আবারো, কনভলিউশন এবং পুলিং লেয়ারের সাহায্যে যতটা সম্ভব সিমপ্লিফ্যায়েড কিন্তু অর্থবহ ইমেজে নিয়ে আসা হয়। এরপর সেই লেয়ারের আউটপুট কে ফুলি কানেক্টেড এক বা একাধিক লেয়ারে ইনপুট হিসেবে দিয়ে সবার সাথে সবার গুন/যোগ করে স্কোর জেনারেট করা হয়। ভ্রমণটা ট্রেনিং টাইপের হলে স্কোর এবং আসল আউটপুট এর পার্থক্য দেখে চক্কর দিতে থাকে এরর কমানোর জন্য। আর ভ্রমণটা ট্রেনিং শেষে প্রেডিকশনের জন্য হলে, একটা স্কোর দিয়ে দেয় যার মাধ্যমে আমরা চিনতে পারি যে ফটোটা নৌকার।

বিভিন্ন ডিপ লার্নিং লাইব্রেরীর ব্যবহার

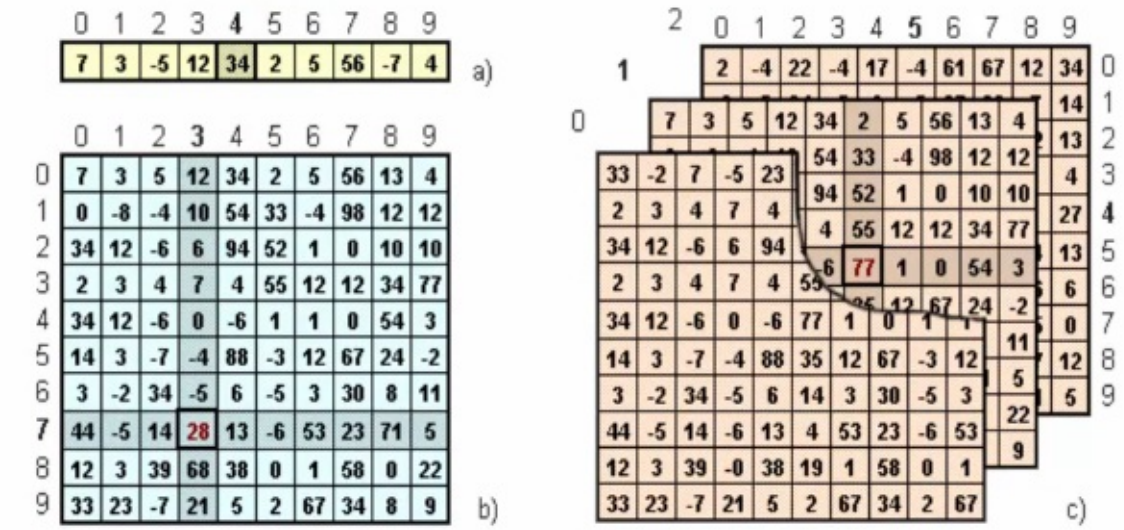
আর্টিফিশিয়াল নিউরাল নেটওয়ার্কের গভীরের কর্মকাণ্ড গুলোর উপর যথেষ্ট পরিমাণ স্বচ্ছ ধারনা থাকার পরেই কেবল এরকম কিছু লাইব্রেরীর ব্যবহার করা উচিৎ হবে। TensorFlow বা সিমিলার লাইব্রেরী গুলো আসলে কি করে? আপনি ম্যানুয়ালি হয়ত Matrix Multiplication, Activation Function, Cost Function বা Gradient Descent করার জন্য এবং নিউরাল নেটওয়ার্কের লেয়ার বা নিউরন গুলো ডিফাইন করার জন্য এক গাদা কোড লিখবেন। যেভাবে এখন পর্যন্ত আমরা করেছি। আবার হয়ত, ইমেজ বা সাইন্ড নিয়ে কাজ করার সময় প্রাথমিক স্টেজে ইমেজ/সাউন্ড ডাটা ম্যানিপুলেট করার জন্যও কিছু কোড লিখবেন, যেমন - ইমেজকে গ্রেস্কেলে কনভার্ট করা, চ্যানেল আলাদা করা, ডাইমেনশন ঠিক ঠাক করা ইত্যাদি। মেশিন লার্নিং এবং ডিপ লার্নিং লাইব্রেরী গুলো বস্তুত এই ম্যানুয়াল কাজ গুলোর জন্যই কিছু রেডিমেড ফাংশন বা মেথড বানিয়ে রেখেছে। অর্থাৎ যে কাজ গুলো সবসময়ই করতে হয় যেকোনো মডেল নিয়ে কাজ করার সময়, সেগুলোর জন্য বিভিন্ন হেল্পার ফাংশন এবং আরও কিছু উপকারী ও গুরুত্বপূর্ণ ফিচার নিয়েই এরকম লাইব্রেরী গুলো তৈরি।</p>

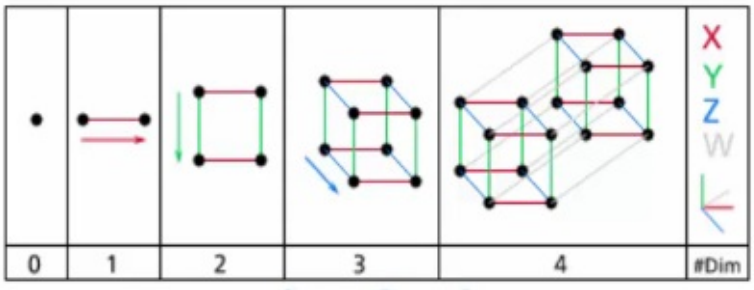
এগুলোকে হাই লেভেল বলতে হবে কারন আপনি অনেক ফাংশনের শুধু নাম ব্যবহার করেই সেটার ইমপ্লিমেন্টেশন করে ফেলতে পারেন। সেগুলোর গভীরে আসলেই কি ম্যাথ বা লজিক কাজ করছে তা জানা লাগবে না। এমনকি TensorFlow, Theano টাইপের হাই লেভেল লাইব্রেরীর উপর ভিত্তি করেও আরও হাই লেভেল লাইব্রেরী তৈরি হচ্ছে, যেমন - Keras. যত হাই লেভেল, তত বেশি অ্যাবসট্র্যাক্ট লেয়ার অর্থাৎ আপনার কাছ থেকে তত বেশি কর্মকাণ্ড হাইড করে রাখা। তাই, সবসময় হাই লেভেল লাইব্রেরী ব্যবহারে অভ্যস্ত হওয়া উচিৎ নয়। এটা সব রকম কন্টেক্সট এর জন্যই মেনে চলা উচিৎ। সেটা মেশিন লার্নিং হোক বা সাধারণ প্রোগ্রামিং বা ডেভেলপমেন্ট হোক।

যাই হোক TensorFlow কিন্তু Google এর ডেভেলপ করা লাইব্রেরী আর তাই আমি অন্তত এটাকে সুদূরপ্রসারী মনে করি। C/C++ আছে এই লাইব্রেরীর ব্যাকএন্ডে আর ফ্রন্টএন্ড ইন্টারফেইস আছে পাইথন এবং অন্য ল্যাঙ্গুয়েজের জন্য। জটিল নিউমেরিক্যাল ক্যালকুলেশন সম্পন্ন টাস্কগুলোকে সহজ এবং টাইম ইফিসিয়েন্ট করার জন্যই এই লাইব্রেরীর আবির্ভাব। ন্যাটিভ পাইথনে যার একটু ল্যাগ ছিল। আর এই লাইব্রেরী কাজ করে Data Flow Graph স্টাইলে অর্থাৎ ম্যাথেম্যাটিক্যাল অপারেশন নিয়ে নোড (Node) এবং মাল্টিডাইমেনশনাল অ্যারে নিয়ে এইজ (Edge) -এর সমন্বয়ে একটি ডাটা ফ্লো গ্রাফ তৈরি হয়।

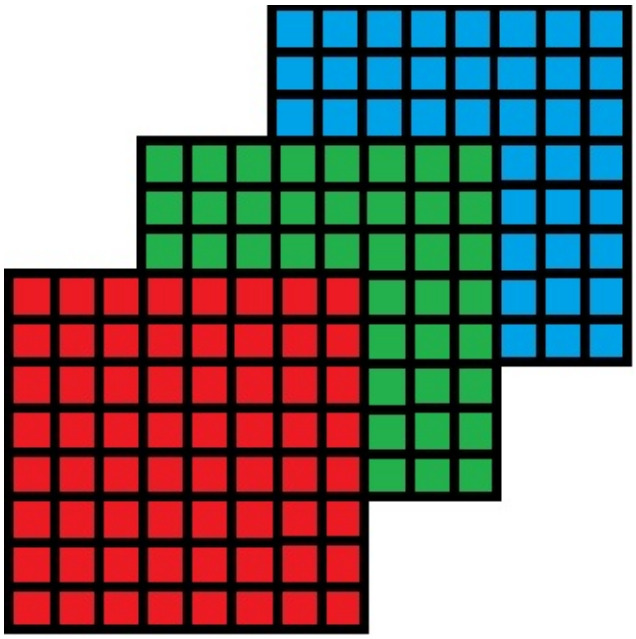
> আপনি কি আগে জানতেন - বিশেষ ধরনের মাল্টিডাইমেনশনাল অ্যারে টাইপের ডাটা স্ট্রাকচারকেই Tensor বলে? :)

নিচের ফিগার দুটো দেখলে আর সাধারণ জ্যামিতির ধারনা থাকলেই বুঝতে পারবেন Tensor মানে কোন ধরনের স্পেশাল ডাটা অবজেক্ট।





0 ডাইমেনশন মানে কোন ভ্যালুর একটাই অবস্থা/অস্তিত্ব যেমন একটা বিন্দু। 1 ডাইমেনশন মানে শুধু একদিকে গমন করে এরকম কিছু ডাটা পয়েন্ট। যেমন আমাদের অতি পরিচিত সাধারণ একটি 1D অ্যারে - [5, 10, 15, 20] এরকম. 2D ডাইমেনশনের উদাহরণ হতে পারে একটি ম্যাট্রিক্স, যেমন - একটা গ্রে-স্কেল ইমেজের পিক্সেল রিপ্রেজেন্টেশন। সেটা আমরা যেকোনো প্রোগ্রামিং ল্যাঙ্গুয়েজেই মাঝে মধ্যে ইমপ্লিমেন্ট করে থাকি এভাবে - [[210 100 255], [200 150 160], [210 0 100]]. 3D এর উদাহরণ হতে পারে একটি কালার ফটোর ম্যাট্রিক্স রিপ্রেজেন্টশন যেখানে Red, Green, Blue প্রত্যেকটি কালারের জন্য তিনটি আলাদা আলাদা 2D ম্যাট্রিক্স থাকে এবং সেগুলো একটার উপর আরেকটা বসিয়ে একটি অবস্থানের তিনটি আলাদা ম্যাট্রিক্সের সেল ভ্যালু হিসাব করে কালার তৈরি করে (নিচের মত),



4D এর উদাহরণ হতে পারে যখন একটি 3D ডাটা অবজেক্ট সময়ের সাথে পরিবর্তন হয়। সেই ডাইমেনশনটাকে TimeSpace ডাইমেনশনও বলা হয়ে থাকে।

আমরা TensorFlow দিয়ে উপরে আলোচিত কয়েকটি ডাইমেনশনের ডাটা অবজেক্টকে তৈরি করতে পারি (কোড এখন না বুঝলেও একটু পরেই বুঝতে পারবেন),

```
import tensorflow as tf

scalar = tf.constant([2])
vector = tf.constant([3, 4, 5])
matrix = tf.constant([[6, 7, 8], [9, 10, 11], [12, 13, 14]])
tensor = tf.constant([ [[6, 7, 8], [9, 10, 11], [12, 13, 14]], [[15, 16, 17], [18, 19,
20], [21, 22, 23]] , [[15, 16, 17], [18, 19, 20], [21, 22, 23]] ])

with tf.Session() as session:
    result = session.run(scalar)
    print("Scalar Data Example:\n", result)

    result = session.run(vector)
    print("\nVector Data:\n", result)

    result = session.run(matrix)
    print("\nMatrix:\n", result)

    result = session.run(tensor)
    print("\nTensor:\n", result)
```

আউটপুট,

```
Scalar Data Example:
 [2]

Vector Data:
 [3 4 5]

Matrix:
 [[ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

Tensor:
 [[[ 6  7  8]
  [ 9 10 11]
  [12 13 14]]

 [[15 16 17]
  [18 19 20]
  [21 22 23]]

 [[15 16 17]
  [18 19 20]
  [21 22 23]]]
```

যা হোক, TensorFlow নিয়ে কাজ করার ধাপ দুইটা - উপরে উল্লেখিত স্টাইলে গ্রাফ তৈরি এবং তারপর সেই গ্রাফকে রান বা এক্সিকিউট করা। আবার বলি, গ্রাফের মধ্যে থাকে কিছু ডিফাইন করা অপারেশন।

হ্যালো ওয়ার্ল্ড

যেহেতু এটা পাইথন লাইব্রেরী ব্যতীত আর কিছুই না। তাই এর ইন্সটলেশন আর দশটা স্বাভাবিক পাইথন লাইব্রেরীর মতই। অর্থাৎ, আপনার পছন্দের মেশিনে বা পাইথন রিয়েল/ভার্চুয়াল এনভায়রনমেন্টে pip ইন্সটলার দিয়ে সহজেই ইন্সটল করে নিতে পারেন নিচের কমান্ড ইস্যু। করে।

```
pip3 install --upgrade tensorflow #I only know what modern Python is. No idea what 2 v
s 3 means.
```

এর ইন্সটলেশন নিয়ে অযথাই প্যাঁচানোর মানে হয় না। যেখানে ইচ্ছা এই লাইব্রেরী ইন্সটল দিবেন। Anaconda, Miniconda হাবিযাবি আপাতত ভুলে জান। আপনার কম্পিউটার আছে, সেই কম্পিউটারে পাইথন ইন্সটলড অবস্থায় আছে। সাথে ধরে নিচ্ছি pip ইন্সটল্ড আছে। ব্যস pip দিয়ে TensorFlow ইন্সটল করে নিবেন আর নিচের মত প্রোগ্রাম লিখে রান করতে থাকবেন স্ক্রিপ্ট (বা মডিউল) মুডে।

নিচের প্রোগ্রামে আমরা TensorFlow দিয়ে একটি খুব সহজ কম্পিউটেশন করেছি যাতে আমরা ব্যাসিক অপারেশন এর ধাপটা সম্বন্ধে পরিষ্কার ধারনা পাই।

```
import tensorflow as tf

a = tf.constant([2])
b = tf.constant([3])

c = tf.add(a,b)

session = tf.Session()

result = session.run(c)
print(result)

session.close()
```

প্রথমেই লাইব্রেরীকে ইম্পরট করা হয়েছে। তারপর দুটো Source Operation (যে অপারেশনের বাইরে থেকে ইনপুট দরকার নাই বা Source Ops) ডিফাইন করা হয়েছে। এই ধরনের সোর্স অপ্স, অন্য অপারেশনের কাছে তাদের তথ্য পাঠায় যেখানে মুল কম্পিউটেশনটা ঘটে। এখানে a, b তে দুটো সোর্স অপ্স tf.constant([2]) এবং tf.constant([3]) এর আউটপুট জমা হচ্ছে। এরপরে লাইনে আমরা আরেকটি কম্পিউটেশনাল অপারেশন tf.add(a,b) ডিফাইন করেছি (এটা কিন্তু ইনপুট পায়)।

এরপর আছে Session. এটা জেনে রাখা গুরুত্বপূর্ণ। যখন বিভিন্ন অপারশন মিলিয়ে একটি গ্রাফ ডিজাইন করা হয় তখন সেটার রান করানোর আগে পুরো গ্রাফকে একটি সেশনে স্টোর করতে হয়। সেশনটি সব অপেরাশন গুলোকে ট্রান্সলেট করে এবং যে ডিভাইসে এই কম্পিউটেশনটা ঘটবে তার কাছে পাঠিয়ে দেয়। একারণেই আমরা আমাদের

গ্রাফকে সেশনে স্টোর করার জন্য একটি Session অবজেক্ট তৈরি করছি। এরপর ওই সেশনকে রান করিয়ে আমরা একটু আগে ডিফাইন করা c অপারশনের আউটপুট দেখতে চাচ্ছি। শেষ লাইনের মাধ্যমে সেশনটি ক্লোজ করা হয়। (ফাইল অপারেশনের কথা মনে পরছে?)

বার বার সেশন তৈরি করা এবং কাজ শেষে সেশন ক্লোজ করার ঝামেলা থেকে মুক্তি পেতে আমরা পাইথনের with ব্লকের সাহায্য নিতে পারি নিচের মত,

```
import tensorflow as tf

a = tf.constant([2])
b = tf.constant([3])

c = tf.add(a,b)

with tf.Session() as session:
    result = session.run(c)
    print(result)
```

দুটো প্রোগ্রামের আউটপুট একই আসবে [5]. এ অবস্থায় হয়ত মনে হচ্ছে নর্মাল একটা যোগ করতে এতো কাহিনী? আসলে এটার মাধ্যমে আপনি বুঝতে পারলেন TensorFlow কিভাবে কাজ করে। কিন্তু এই লেভেলের সস্থা কাজ নিশ্চয়ই আপনি TensorFlow দিয়ে করাবেন না, তাই না? অতি জটিল কম্পিউটেশন সহজ করার সাথে সাথে এই লাইব্রেরীর আরও কিছু সুবিধার মধ্যে আছে - আপনি একই গ্রাফ যেকোনো রকম হার্ডওয়্যার প্ল্যাটফর্মে রান করাতে পারবেন। CPU, GPU, Cluster বা Android, iOS. যাই হোক। বলে নেয়া ভালো Google I/O 17 এ Google কিন্তু নতুন এক ধরনের হার্ডওয়্যার TPU (Tensor Processing Unit) রিলিজ দিয়েছে যা স্পেশালি Tensor ভিত্তিক কম্পিউটেশনের জন্য ইফেক্টিভলি ডিজাইন করা।

আরেকবার আমরা TensoFlow এর একটা এক লাইনের সংজ্ঞা লিখে ফেলি - “একটি কম্পিউটেশন গ্রাফে বিভিন্ন অপারেশনের মধ্যে টেনসর ফর্মে ডাটা আদান প্রদান করে একবারে পুরো কম্পিউটেশন যেকোনো হার্ডওয়্যারে সম্পন্ন করার জন্য একটি মডেল বা লাইব্রেরী হচ্ছে TensorFlow.” আর এতো লম্বা মনে না থাকলে শুধু এটুকু মনে রাখুন এটা একটা “কম্পিউটেশনাল লাইব্রেরী” :D

ভ্যারিয়েবল ও প্লেসহোল্ডার

এখন যেহেতু আমরা ডাটা স্ট্রাকচার গুলো নিয়ে মোটামুটি একটা ধারনা পেলাম, চলুন দেখি TensorFlow কিভাবে ভ্যারিয়েবল হ্যান্ডেল করে। ভ্যারিয়েবল ডিফাইন করতে হয় tf.Variable() স্টেটমেন্টের মাধ্যমে। তবে মনে রাখা জরুরি - কম্পিউটেশন গ্রাফে ভ্যারিয়েবল গুলো ব্যবহারের জন্য সেগুলোকে ইনিসিয়ালাইজ করতে হবে (গ্রাফকে সেশনে রান করানোর আগেই)। এটা করা হয় tf. global_variables_initializer() এর মাধ্যমে। কোন ভ্যারিয়েবলের ভ্যালু আপডেট করার জন্য আমাদেরকে assign অপারেশনকে রান করতে হয়।

নিচের প্রোগ্রামটি খেয়াল করি,

```
import tensorflow as tf

state = tf.Variable(0)

one = tf.constant(1)
new_value = tf.add(state, one)
update = tf.assign(state, new_value)

init_op = tf.global_variables_initializer()

with tf.Session() as session:
    session.run(init_op)
    print(session.run(state))
    for _ in range(3):
        session.run(update)
        print(session.run(state))
```

উপরের প্রোগ্রামে, প্রথমেই tensorflow ইম্পোরট করে নেয়া হয়েছে। এরপর একটি ভ্যারিয়েবল state, ডিফাইন এবং ইনিসিয়ালাইজ করা হয়েছে 0 ভ্যালু নিয়ে। এরপর একটি source ops ডিফাইন করা হয়েছে। এরপর একটি কম্পিউটেশনাল অপারেশন, add এর ব্যবহার করা হয়েছে যা মূলত state ভ্যারিয়েবলের সাথে 1 যোগ করে। কিন্তু যেহেতু state ভ্যারিয়েবলের মান এখনো পরিবর্তন (update) হয় নি, তাই সেটা করার জন্য একটি assign অপারশন ডিফাইন করতে হচ্ছে। এরপরের লাইনে, উপরোক্ত ভ্যারিয়েবলকে ইনিসিয়ালাইজ করা হয়েছে কম্পিউটেশন গ্রাফে সেগুলো ব্যবহার করার জন্য। অতঃপর একটি with ব্লকের মধ্যে, সেশন অবজেক্টের মাধ্যমে, প্রথমে ভ্যারিয়েবল ইনিসিয়ালাইজার অপারেশনকে রান করা হয়েছে। এবং state ভ্যারিয়েবলের শুরুর মান প্রিন্ট করা হয়েছে। এটার জন্য স্ক্রিনে প্রথম প্রিন্ট ভ্যালু 0 আসছে। এরপর একটি ফর লুপ চালিয়ে update অপারেশনকে রান করানো হচ্ছে এবং প্রত্যেকবার আপডেট অপারেশনের পর state ভ্যারিয়েবলের মান প্রিন্ট করা হচ্ছে।

আউটপুট,

```
0
1
2
3
```

এবার আসি প্লেস হোল্ডারের প্রসঙ্গে। যদি আপনি TensorFlow মডেলের বাইরে থেকে এটাকে ডাটা ইনপুট/ফিড করাতে চান তাহলে দরকার পরে এই প্লেসহোল্ডার। প্লেস হোল্ডার একরকম ভ্যারিয়েবল যেটাতে আসলে নির্দিষ্ট একটি সময় পর্যন্ত এর আসল কোন ভ্যালু থাকে না শুধু স্ট্রাকচার থাকে। একটু অন্যভাবে বলতে গেলে, প্লেসহোল্ডার মানে আপনার মডেলের কিছু গর্ত, যে গর্ত গুলোতে আপনি কিছু পরে ডাটা ঢুকাবেন। প্লেসহোল্ডার তৈরি করতে placeholder মেথড ব্যবহার করতে হয়। তৈরির সময় ডাটা টাইপ এবং এর প্রেসিশন ভ্যালুও ডিফাইন করতে হবে। নিচের টেবিলে প্রত্যেকটি ডাটাটাইপ এবং সে অনুযায়ী সিনট্যাক্স দেয়া আছে,

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.
DT_QUINT8	tf.quint8	8 bits unsigned integer used in quantized Ops.

এবার নিচের প্রোগ্রামটি দেখি,

```
import tensorflow as tf

a = tf.placeholder(tf.float32)
b = a * 2

with tf.Session() as session:
    result = session.run(b)
    print(result)
```

এখানে প্রথমে একটি প্লেসহোল্ডার ডিফাইন করা হয়েছে float টাইপের। অতঃপর সেই প্লেসহোল্ডার ভ্যারিয়েবলের সাথে 2 গুন করার অপারশন ডিফাইন করা হয়েছে b তে (স্বাভাবিক যোগ বিয়োগের ক্ষেত্রে tf এর মেথড ব্যবহার না করে, স্বাভাবিক অপারেটর ব্যবহার করলেও চলে)। এবং স্বাভাবিক ভাবে session এর রান মেথডের মধ্যে b কে চালিয়ে দিয়ে এই অপারশণকে রান করার কথা বলা হচ্ছে। সাথে result প্রিন্ট এর কথা বলা হচ্ছে। কিন্তু এই প্রোগ্রাম এরর থ্রো করবে (যেমন- You must feed a value for placeholder tensor), কারন প্লেসহোল্ডার ডিফাইন করলেও গ্রাফ রান করানোর সময় আমরা সেই প্লেসহোল্ডারের জন্য আসল কোন ভ্যালু গ্রাফ/মডেলের মধ্যে পাঠাই নি। পাঠানোর

কাজটা খুবি সহজ, run মেথডের আরেকটি আর্গুমেন্ট হিসেবে আমরা প্লেসহোল্ডারের আসল ভ্যালু পাস করে দিতে পারি নিচের মত। পাস করার সময় প্লেসহোল্ডারের নাম এবং করেস্পন্ডীং ভ্যালুকে পাইথন ডিকশনারি ফরম্যাটে পাঠাতে হবে।

```
import tensorflow as tf

a = tf.placeholder(tf.float32)
b = a * 2

with tf.Session() as session:
    result = session.run(b, feed_dict={a:3.5})
    print(result)
```

আউটপুট আসবে 7.0

যেহেতু TensorFlow তে আমরা মাল্টি ডাইমেনশনাল ডাটা পাস করতে পারি। তাই চাইলে আমরা a এর ভ্যালু হিসেবে একটি Tensor -ও পাঠিয়ে দেখতে পারি। সেই টেনসরের উপর আমাদের ডিফাইন করা গুন a*2 -ই অ্যাপ্লাই হবে। নিচের প্রোগ্রাম দেখুন,

```
import tensorflow as tf

a = tf.placeholder(tf.float32)
b = a * 2

with tf.Session() as session:
    result = session.run(b, feed_dict={a:[ [[6, 7, 8], [9, 10, 11], [12, 13, 14]], [[15
, 16, 17], [18, 19, 20], [21, 22, 23]] , [[15, 16, 17], [18, 19, 20], [21, 22, 23]] ]}
)
    print(result)
```

আউটপুট,

```
[[[ 12.  14.  16.]
  [ 18.  20.  22.]
  [ 24.  26.  28.]]

 [[ 30.  32.  34.]
  [ 36.  38.  40.]
  [ 42.  44.  46.]]

 [[ 30.  32.  34.]
  [ 36.  38.  40.]
  [ 42.  44.  46.]]]
```

TensorFlow দিয়ে ইমেজ ক্লাসিফিকেশনের উপযোগী একটি NN তৈরি

এই সেকশনে আমরা, অনেক রকম মানুষের বিভিন্ন রকম হাতের লেখা ওয়ালা কিছু নাম্বার/ডিজিট এর ফটো কালেকশন দিয়ে একটা নিউরাল নেটওয়ার্ক-কে ট্রেইন করিয়ে তারপর কিছু টেস্ট ফটো দিয়ে সেগুলোর সঠিক ক্লাসিফিকেশন জানার চেষ্টা করবো। সহজ ভাবে বলতে - "হ্যান্ড রিটেন ডিজিট ক্লাসিফিকেশন প্রব্লেম"।

এই টিউটোরিয়ালে আমরা কনভলিউশনাল নিউরাল নেটওয়ার্ক মডেল ব্যবহার করছি না। বরং সিঙ্গেল লেয়ারের লিনিয়ার মডেল ব্যবহার করবো। অর্থাৎ একটি ইনপুট লেয়ার এবং একটি আউটপুট লেয়ার থাকবে, কিন্তু ইনপুট লেয়ারে প্রথম দিকের উদাহরণ এর মত কয়েকটি না বরং অনেক গুলো নিউরন থাকবে। আর আউটপুট লেয়ারে থাকবে ১০টি নিউরন। ১০ ধরনের ডিজিট ক্লাসিফিকেশনের জন্য। এতে করে আমাদের TensorFlow দিয়ে কাজ করার কমন কিছু স্টেপ সম্বন্ধে পরিষ্কার ধারনা আসবে। তবে হ্যাঁ, এই লিনিয়ার ক্লাসিফায়ারও যথেষ্ট ভালো মতই ডিজিট ক্লাসিফিকেশন করতে পারবে আশা করা যায়। অন্তত ৮৫-৯০% সঠিক ক্লাসিফাই করতে পারবে। এই টিউটোরিয়ালের পর আমরা একই সমস্যা আরও ইফেক্টিভ ভাবে সমাধানের জন্য এবং অ্যাকিউরেসি লেভেল আরও বাড়ানোর জন্য কনভলিউশনাল নিউরাল নেটওয়ার্ক মডেল ব্যবহার করবো। যা হোক, এখন আমরা ধাপে ধাপে সব গুলো কাজ করবো এবং লেয়ার তৈরি করবো এবং তার মাঝে মাঝেই কিছু নতুন টার্ম আসবে, সেগুলোর প্রয়োজনীয়তা এবং সংজ্ঞা জানবো।</p> প্রথমেই আমরা কিছু রেডিমেড ডাটা সেট নিয়ে কাজ করবো। অর্থাৎ ডিজিট ক্লাসিফিকেশন শিখতে গিয়ে আমাদের অনেক অনেক ইমেজ দরকার পরবে আমাদের মডেলকে ট্রেনিং দেয়ার জন্য, তাই না? এমনকি ট্রেনিং ডাটাগুলো লেবেল্ড (কোন ফটো কোন ডিজিট তার একটা ম্যাপিং) হতে হবে। নাহলে ট্রেনিং হবে ক্যামনে? এখন নিউরাল নেটওয়ার্ক শিখতে গিয়ে যদি মাসের পর মাস সময় দিয়ে শুধু ডাটাই রেডি করতে হয় তাহলে ক্যামনে কি? মজার বিষয় হচ্ছে, TensorFlow -এর সাথেই এরকম কিছু রেডিমেড ডাটা থাকে এবং যেগুলো চাইলে আমরা import করে সেগুলোর উপর কাজ করতে পারি। অন্তত আসল জিনিষ শেখার সময় আমাদের পুরো সময়টা ডাটা প্রি-প্রসেসিং -এ নষ্ট হচ্ছে না। ফোকাস থাকবে মডেল ডেভেলপমেন্টে। যা হোক, এই ডাটাবেজটার নাম হচ্ছে MNIST ডাটাবেজ/ডাটাসেট।

এই টিউটোরিয়ালের জন্য আমরা Jupyter Notebook ব্যবহার করবো। এতে করে ধাপে ধাপে আলোচনা করে করে আগানো যাবে এবং আগের ধাপে রান করা কোড পরের ধাপেও অ্যাক্সেস করা যাবে। jupyter Notebook সম্পর্কে ধারনা না থাকলে একটু অন্য কোথাও থেকে আপাতত দেখে আসতে পারেন। এটা তেমন কিছু না। একটা ওয়েব অ্যাপ। এতে করে ব্রাইজারের মধ্যে একটা পেজে কোড এবং বাংলা ইংলিশ মিলিয়ে লেখা যায় এবং কোড গুলোকে রানও করা যায়। আর পুরো ডকুমেন্টের রানটাইম একটাই থাকে। এটাও খুব সহজে প্যাকেজ আকারেই ইন্সটল করা যায় এবং একটা কমান্ড দিয়েই রান করানো যায়। কথা না বাড়িয়ে শুরু করা যাক।

যদিও শেষের দিকে পুরো প্রোগ্রামের একটা স্ক্রিপ্ট ভার্সন থাকবে যেটা স্বাভাবিকভাবে নোটবুকের বাইরেও রান করানো যাবে।

```
# Cell 1
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
from sklearn.metrics import confusion_matrix
```

উপরের ব্লক নিয়ে কিছু বলার দরকার আছে কি? সব চেনা জিনিষ, একটা বাদে। confusion_matrix ব্যবহার করে আমরা একধরনের স্পেশাল ম্যাট্রিক্স তৈরি ও ডিসপ্লে করতে পারি যার মাধ্যমে আমরা কিছু রিলেটেড এরর এর বৈশিষ্ট্য সম্পর্কে একটা ভিজুয়াল ধারনা পাবো। এটার স্টেপ আসা মাত্রই এর দরকারটাও বোঝা যাবে। যদিও এটা অপশনাল স্টেপ। আমাদের মুল মডেল তৈরিতে এটার গুরুত্ব নাই, বরং Accuracy বাড়াতে এবং সমস্যার উপর একটা স্পষ্ট ধারনা আনতে সাহায্য করবে এই ম্যাট্রিক্স। অর্থাৎ, সমস্যা নিয়ে ভালো অব্জারভেশন করতে চাইলে এগুলো লাগে। আরেকটা জিনিষ - %matplotlib inline যার মাধ্যমে জুপিটার নোটবুকের চলতি ডকুমেন্টটির মধ্যেই প্লটিং গুলো ডিসপ্লে করার কথা বলা হচ্ছে। তো নোটবুকের প্রথম সেলে এই কোড লিখে সেলটি এক্সিকিউট করে ফেলি।

এরপর আমাদের ডাটাগুলোকে লোড করতে হবে, এর জন্য নিচের কোড টুকু ব্যবহার করতে পারি অর্থাৎ পরের সেলে লিখে সেলটি এক্সিকিউট করতে পারি,

```
# Cell 2
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets("data/MNIST/", one_hot=True)
```

এর মাধ্যমে ১২ মেগাবাইট সাইজের ডাটাসেটটি ডাউনলোড হবে যদি data/MNIST/ পাথে আগে থেকেই ডাটাসেটটি না থাকে।

ডাটা বুঝে নেয়া

পুরো ডাটাসেটে যথাযথ ক্লাস (কোনটা কোন ডিজিট) ম্যাপ করা ৭০০০০ ইমেজ আছে যার মধ্যে ৫৫০০০ হচ্ছে ট্রেনিং ইমেজ, ১০০০০ হচ্ছে টেস্ট ইমেজ এবং ৫০০০ হচ্ছে ভ্যালিডেশন ইমেজ। অর্থাৎ পুরো ডাটাসেটটি ৩টি সাবসেটে বিভক্ত। কিছু ডাটা ট্রেনিং এর জন্য, কিছু ডাটা ভ্যালিডেশনের জন্য, আর কিছু ডাটা হচ্ছে ফাইনাল মডেলকে টেস্ট করার জন্য। এই সাবসেট গুলো মিউচুয়ালি এক্সকুসিভ অর্থাৎ একটি সেটের ডাটা আরেকটি সেটের মধ্যে নাই। অর্থাৎ কমন কোন এলিমেন্ট এই ৩টি সেটের মধ্যে নাই। পরীক্ষা করে দেখতে পারি নিচের কোড ওয়ালা সেলটি এক্সিকিউট করে,

```
# Cell 3
print("Size of:")
print("- Training-set:\t\t{}".format(len(data.train.labels)))
print("- Test-set:\t\t{}".format(len(data.test.labels)))
print("- Validation-set:\t{}".format(len(data.validation.labels)))
```

আউটপুট,

```
Size of:
- Training-set:     55000
- Test-set:         10000
- Validation-set:   5000
```

এই টিউটোরিয়ালে আমরা ভ্যালিডেশন সেটের ব্যবহার করবো না। যা হোক, Cell 2 এর কোডের read_data_sets মেথডের দ্বিতীয় প্যারামিটার নিয়ে একটু কথা বলি. one_hot=True পাঠিয়ে আমরা বলছি যে এই ডাটাসেট এর লেবেল (ফটোর সাপেক্ষে সঠিক উত্তর/ডিজিট) গুলোকে আমরা এই ফরম্যাটে চাই। এই ফরম্যাট ডেসিম্যাল ডিজিটের বাইনারি রিপ্রেজেন্টেশনের মতই কিন্তু একটু অন্যভাবে রিপ্রেজেন্ট করে। মাত্র একটি বিট কে হাই বা 1 করে সেই ডিজিটের অবস্থান প্রকাশ করা হয়। নিচের উদাহরণ দেখলেই ব্যাপারটি সহজেই বোঝা যাবে। যেমন 0 এবং 5 এর বাইনারি রিপ্রেজেন্টেশন হয় নিচের মত,

```
Number representation:    0
Binary encoding:        [2^5]  [2^4]  [2^3]  [2^2]  [2^1]  [2^0]
Array/vector:             0      0      0      0      0      0

Number representation:    5
Binary encoding:        [2^5]  [2^4]  [2^3]  [2^2]  [2^1]  [2^0]
Array/vector:             0      0      0      1      0      1
```

আর One-Hot Vector প্রেজেন্টেশন হয় নিচের মত,

```
Number representation:   0
One-hot encoding:        [5]   [4]   [3]   [2]   [1]   [0]
Array/vector:            0     0     0     0     0     1

Number representation:   5
One-hot encoding:        [5]   [4]   [3]   [2]   [1]   [0]
Array/vector:            1     0     0     0     0     0
```

অর্থাৎ ডিজিটটি যদি 5 হয় তাহলে ৫টি বিট ওয়ালা একটি ভেক্টরের ৫নাম্বার বিটটি হাই অর্থাৎ 1 সেট করে দেয়া হয়। তো, আমাদের আলোচনায় ডাউনলোড করা হাতের লেখার ফটো গুলোর লেবেল গুলো আসছে এই ফরম্যাটে। আমরা ডাটাসেট থেকে প্রথম ৫টি ফটোর লেবেল গুলোর One-Hot Vector রিপ্রেজেন্টেশন দেখতে পারি নিচের মত করে,

```
# Cell 4
data.test.labels[0:5, :]
```

আউটপুট আসবে, নিচের মত,

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.]])
```

তাহলে আমরা দেখে দেখেই বলে দিতে পারি প্রথম ৫টি ডিজিটের লেবেল বা নাম কি। প্রথমটার ৭নাম্বার বিটটি হাই, তাই এটি 7. দ্বিতীয়টির ২ নাম্বার বিট হাই, অর্থাৎ এটি 2 লেখা একটি ফটোর লেবেল/নাম।

আমরা চাইলে একই কাজটা কোড লিখেও করতে পারি। যেমন, নিচের লাইন খেয়াল করুন,

```
# Cell 5
data.test.cls = np.array([label.argmax() for label in data.test.labels])
```

এখানে লুপ চালিয়ে প্রত্যেকটি লেবেল ভেক্টরকে নিয়ে তার উপর argmax() মেথডটি অ্যাপ্লাই করা হয়েছে। এই মেথডের কাজ হচ্ছে একটি ভেক্টরের মধ্যে যে বিটটি হাই থাকবে তার ইনডেক্স রিটার্ন করবে। হয়ে গেলো? আমরা লেবেল গুলোর One-Hot Vector টাইপের রিপ্রেজেন্টেসন থেকে খুব সহজেই সঠিক ডিজিট নাম্বারটা পেতে পারি। এই পুরো কনভার্সনটা একটা numpy array তে কনভার্ট করে স্টোর করা হচ্ছে।

এখন যদি আমরা data.test.cls ভ্যারিয়েবলের প্রথম ৫টি এলিমেন্ট দেখি তাহলে নিচের মত আউটপুট পাবো,

```
# Cell 6
data.test.cls[0:5]
```

```
array([7, 2, 1, 0, 4])
```

এতক্ষণে One-Hot Vector প্রেজেন্টেশন এবং argmax মেথডের কাজ বোঝা গেছে নিশ্চয়ই?

ডাটা ডাইমেনশন

ডাটা ডাইমেনশন সম্পর্কে স্বচ্ছ ধারনা রাখতে হবে মাথায়। কোন ম্যাট্রিক্স বা টেনসরের ডাইমেনশন এর প্রসঙ্গ আসা মাত্রই যাতে কল্পনায় স্পষ্ট একটা ভিউ আসে ওই ডাটা অবজেক্টটার। তাহলে সব কিছু সহজ মনে হবে। যাই হোক, এরকম কিছু ডাইমেনশনকে আমরা কিছু ভ্যারিয়েবলে স্টোর করি এবার এবং সেলটি এক্সিকিউট করে নেই,

```
# Cell 7
# We know that MNIST images are 28 pixels in each dimension.
img_size = 28

# Images are stored in one-dimensional arrays of this length.
img_size_flat = img_size * img_size

# Tuple with height and width of images used to reshape arrays.
img_shape = (img_size, img_size)

# Number of classes, one class for each of 10 digits.
num_classes = 10
```

img_size ভ্যারিয়েবলে আমাদের আলোচ্য ফটোগুলোর ডাইমেনশন স্টোর করছি। MNIST ডাটাসেটের ফটো গুলো 28x28 সাইজের ফটো। আসলে ইমেজের কন্টেক্সট থেকে বলতে, 28x28x1 সাইজের অর্থাৎ ফটো গুলো সাদা কালো এবং এর কালার চ্যানেল একটাই। রঙ্গিন ফটো হলে এদের ডাইমেনশন হত 28x28x3। RGB তিনটা রঙের তিনটা চ্যানেল এবং প্রত্যেক চ্যানেলের জন্য 28x28 সাইজের একগাদা পিক্সেল ভ্যালু। যা হোক, দ্বিতীয় ভ্যারিয়েবলে আমরা প্রত্যেকটি ইমেজের ফ্ল্যাট রিপ্রেজেন্টশন স্টোর করছি অর্থাৎ 28x28 সাইজের একটি ফটোর সবগুলো পিক্সেলকে যদি স্টোর করতে চাই তাহলে আমাদের img_size * img_size সাইজের একটি ওয়ান ডাইমেনশনাল অ্যারে বা ভেক্টর লাগবে। একটি টাপলে ইমেজের সেইপকে স্টোর করছি। আর শেষের ভ্যারিয়েবলে স্টোর করছি আমাদের যতগুলো আউটপুট ক্লাস দরকার সেই সংখ্যাটা। আমাদের ১০ ধরনের ক্লাসিফিকেশন দরকার, কারন ১০টাই ডিজিট দুনিয়াতে।

এ অবস্থায় একটু খুত খুতে লাগতে পারে এটা ভেবে যে - এইযে ফটো গুলো ইম্পরট করলাম এবং সেগুলোর উপর নাকি আবার কাজ করবো। সেগুলো আসলে দেখতে কেমন? ক্লিয়ার ভিউ তো লাগবে নাকি? ;)

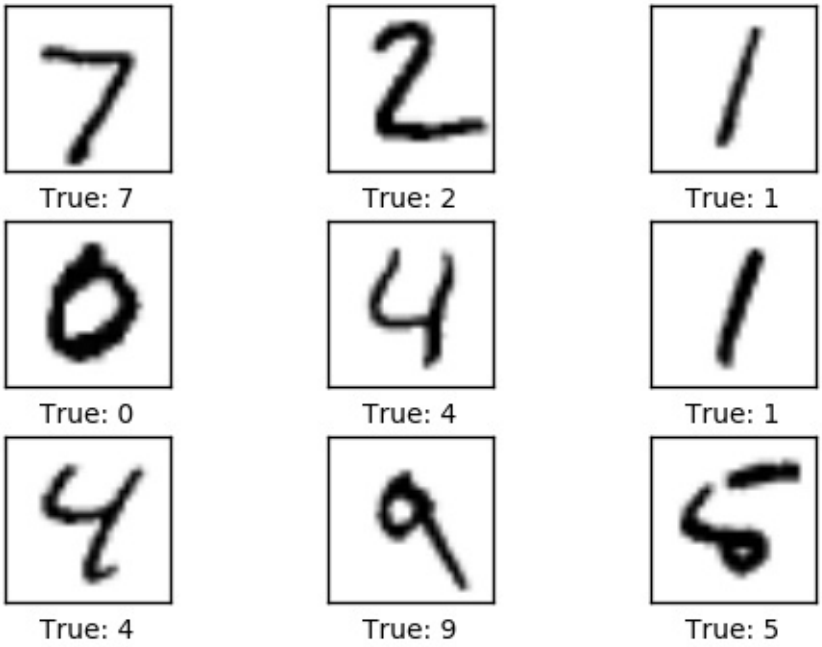
নিচের কোড ব্লকটি পুরো একটি হেল্পার ফাংশন যার মাধ্যমে আমরা MNIST ডাটাসেটের ইমেজ গুলোকে রেন্ডার বা ডিসপ্লে করতে পারবো যেকোনো সময়। এখানে একটি 3x3 গ্রিডে মোট ৯টি ফটো এবং সেগুলোর সঠিক লেবেল ডিসপ্লে করানোর ব্যবস্থা করা হয়েছে।

```
# Cell 8
def plot_images(images, cls_true, cls_pred=None):
    assert len(images) == len(cls_true) == 9

    # Create figure with 3x3 sub-plots.
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_shape), cmap='binary')

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i], cls_pred[i])

        ax.set_xlabel(xlabel)

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])
```

তো, উপরের এই ফাংশনকে কাজে লাগিয়ে আমরা কিছু ফটো এবং সেগুলোর সাপেক্ষে সঠিক লেবেল রেন্ডার করে দেখি,

```
# Cell 9
# Get the first images from the test-set.
images = data.test.images[0:9]

# Get the true classes for those images.
cls_true = data.test.cls[0:9]

# Plot the images and labels using our helper-function above.
plot_images(images=images, cls_true=cls_true)
```

এই অবস্থায় নোটবুকের সেলটি এক্সিকিউট করলে নিচের মত আউটপুট আসবে,


True: 7
True: 2
True: 1
True: 0
True: 4
True: 1
True: 4
True: 9
True: 5

TensorFlow গ্রাফ তৈরি

আগেই বলা হয়েছে, TensorFlow দিয়ে কাজ করতে হলে প্রথমেই একটি পুরনাঙ্গ কম্পিউটেশনাল গ্রাফ তৈরি করতে হয়। এরপর পুরো গ্রাফকে এক সাথে এক্সিকিউট করা যায়। এতে করে পাইথনে আলাদা আলাদা করে ক্যালকুলেশন গুলো লিখে এক্সিকিউট করালে যেমন টাইম বা অন্য কমপ্লেক্সিটি হতে পারতো, তার চেয়ে অনেক ইফেক্টিভলি ক্যালকুলেশন গুলো হয়। গ্রাফে আরও সুবিধা হচ্ছে স্বয়ংক্রিয় ভাবে গ্র্যাডিয়েন্ট ডিসেন্ট বের করা যায় যাতে করে মডেলের ভ্যারিয়েবল (ওয়েট, বায়াস) গুলোকে অপ্টিমাইজ করা যায় সহজেই। যেহেতু পুরো গ্রাফ জুড়ে অনেক গুলো কম্পিউটেশন থাকে এতে করে ডেরিভ্যাটিভ এর চেইন রুল এর মাধ্যমে খুব দ্রুত এবং সহজে পুরো গ্রাফের গ্র্যাডিয়েন্ট ডিসেন্ট বের করা যায়।

এ অবস্থায় আমরা আবার জানবো - একটা TensorFlow গ্রাফ মূলত কি কি নিয়ে গঠিতঃ
১) প্লেসহোল্ডার ভ্যারিয়েবল - যার মাধ্যমে গ্রাফে ইনপুট দেয়া হয়
২) মডেল ভ্যারিয়েবল - ওয়েট, বায়াস ইত্যাদি; মূলত ট্রেনিং করিয়ে এগুলোকে অপ্টিমাইজ করেই একটা ইফিসিয়েন্ট মডেল তৈরি করা হয়
৩) মডেল - সহজ কথায় একটি ম্যাথেম্যাটিক্যাল ফাংশন যেখানে প্লেসহোল্ডারের মাধ্যমে ইনপুট দিয়ে এবং মডেল ভ্যারিয়েবলের সমন্বয়ে আউপুটপুট পাওয়া যায়
৪) *Cost* - এই মানের উপর ভিত্তি করে মডেল ভ্যারিয়েবল গুলোর অপটিমাইজেশনের দিক নির্দেশনা দেয়া হয়
৫) অপটিমাইজেশন মেথড - এই মেথড মূলত *Cost* কে মাথায় রেখে মডেল ভ্যারিয়েবল গুলোকে আপডেট করে।

প্লেস হোল্ডার তৈরিঃ
আগেও একবার বলা হয়েছে - এর মাধ্যমে গ্রাফে ইনপুট দেয়া হয়। যেমন, প্লেসহোল্ডারকে মেনে এক এক বার এক এক সেট ইনপুট দেয়া যাবে গ্রাফে। অনেকে বলেন গ্রাফকে ফিড করানো। তো, আমরা আসলে গ্রাফে কি ইনপুট দিবো? ইমেজ বা হাতের লেখা ওয়ালা ফটো গুলোকে, তাই তো? আমরা প্রথমবার হয়ত ১০০টা ইমেজ গ্রাফে ইনপুট দিলাম। পরেরবার আরও ২০০ দিলাম। তাই, ইমেজ ইনপুট দেয়ার জন্য একটা প্লেসহোল্ডার ভ্যারিয়েবল থাকলে ভালো। এই প্লেসহোল্ডার হবে Tensor টাইপের। Tensor মানে? মাল্টিডাইমেনশনাল ভেক্টর বা ম্যাট্রিক্স :) Tensor এর ডাটাটাইপ হবে float32. আর এর সেইপ হবে [None, img_size_flat] None মানে হচ্ছে এই টেনসরটি যেকোনো সংখ্যক ইমেজ নিতে পারবে যে ইমেজ গুলো কিনা এক একটি ফ্ল্যাট অ্যারে অর্থাৎ সবগুলো পিক্সেলের ফ্ল্যাট ভেক্টর তথা আমাদের একটু আগের স্টেটমেন্ট অনুযায়ী img_size_flat. অর্থাৎ স্টেটমেন্টটি হবে,

```
# Cell 10
x = tf.placeholder(tf.float32, [None, img_size_flat])
```

আরেকবার বলি - যেমন উপরের স্টেটমেন্ট এর প্লেসহোল্ডারে যদি আমরা যেকোনো সময় মাত্র দুটি 28x28 সাইজের ফটোকে ইনপুট হিসেবে দিয়ে গ্রাফ এক্সিকিউট করি তাহলে উপরের স্টেটমেন্টার অভ্যন্তরীণ চেহারা হবে এরকম, x = tf.placeholder(tf.float32, [2, 784]) এবং ডাটার চেহারা হবে [[p00, p01 .... p0783], [p10, p11 .... p1783]]. pxx হচ্ছে পিক্সেল ভ্যালু।

এবার আরও একটা প্লেসহোল্ডার নেবো যেখানে সময় মত ইনপুট দেবো, একটু আগে ইনপুট দেয়া ইমেজ গুলোর সঠিক লেবেল গুলোকে। এই প্লেসহোল্ডার ভ্যারিয়েবলের সেইপ হবে [None, num_classes] টাইপের? কেন? None মানে যেকোনো সংখ্যক লেবেল সেট নিতে পারবে আর প্রত্যেকটা লেবেল সেট হবে num_classes অর্থাৎ 10 লেন্থ এর

ভেক্টর। আগের প্লেসহোল্ডার ভ্যারিয়েবলটির নাম ছিল x এবং এই প্লেসহোল্ডার ভ্যারিয়েবলটির নাম y_true.

```
# Cell 11
y_true = tf.placeholder(tf.float32, [None, num_classes])
```

এবার আমাদের আরেকটি প্লেসহোল্ডার দরকার পরবে। এর মধ্যে দেয়া হবে x প্লেসহোল্ডারের প্রত্যেকটি ইমেজের জন্য এর টু ক্লাস। অর্থাৎ এটার ধরন হবে ইন্টিজার টাইপের। কারন টু ক্লাস গুলো তো (0,1,2,3 ... 9) এরকম. এর সেইপ হবে [None] অর্থাৎ, এই প্লেসহোল্ডারটি একটি ওয়ান ডাইমেনশনাল ভেক্টর কিন্তু যার লেন্থ হতে পারে যেকোনো সংখ্যক। অর্থাৎ একটি ইমেজের ক্ষেত্রে এটি শুধুমাত্র ওই ইমেজেটির টু ক্লাস/লেবেল হোল্ড করবে আবার ৫০টা ইমেজের জন্য ৫০টা টু ক্লাস হোল্ড করবে। এই আর কি,

```
# Cell 12
y_true_cls = tf.placeholder(tf.int64, [None])
```

এখন পর্যন্ত x, y_true এবং y_true_cls এই তিনটা প্লেসহোল্ডার ভ্যারিয়েবলকে মাথার মধ্যে পরিষ্কার ভাবে স্টোর করুন। দরকার হলে এই সেকশনের শুরু থেকে আরেকবার পরে আসুন।

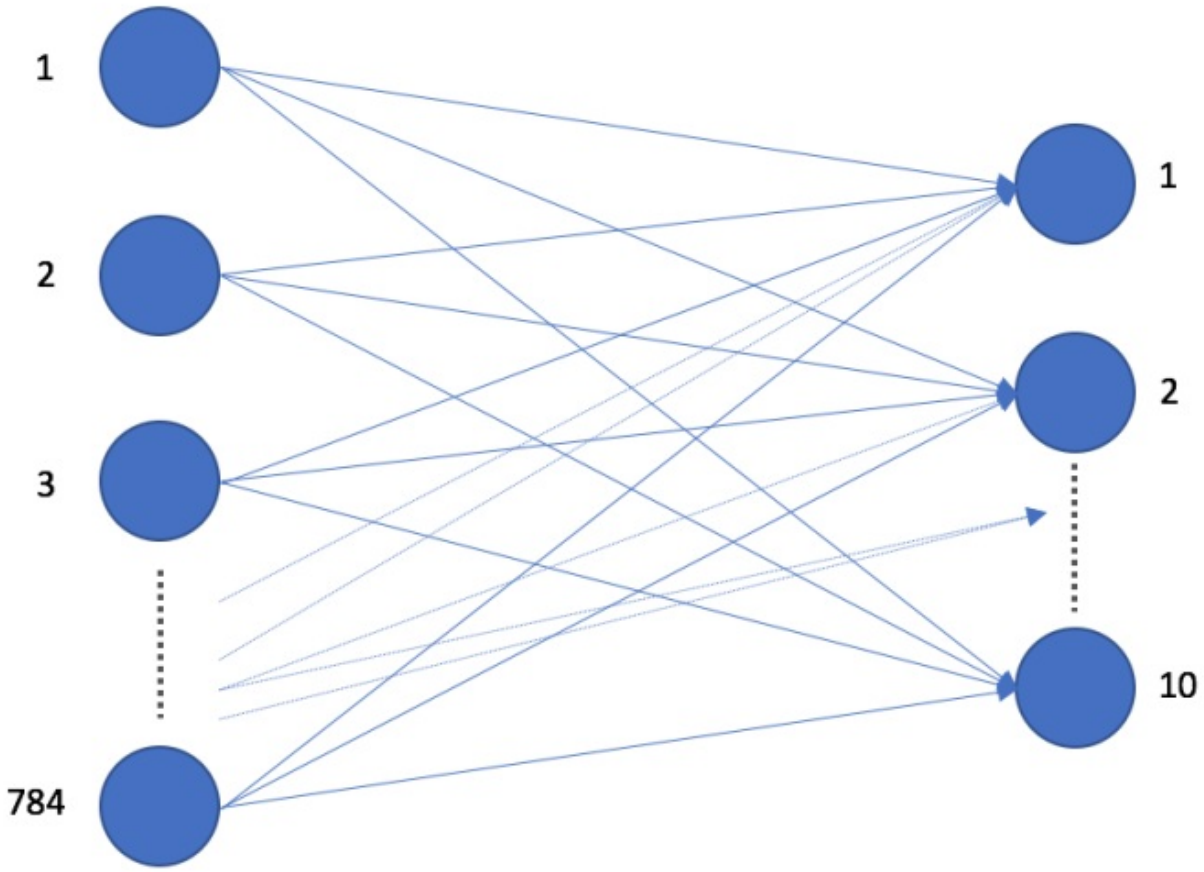
এবার আসি মডেল ভ্যারিয়েবলেঃ
এই পোস্টের একদম শুরুতে যে নিউরাল নেটওয়ার্কের উদাহরণ দেয়া হয়েছে সেটা মনে আছে? ওখানে কিন্তু আমরা ট্রেইন করে করে কিছু ওয়েট ঠিক করেছিলাম যেগুলোর উপর ভিত্তি করেই পরবর্তীতে ওই মডেল নতুন ইনপুট নিয়ে ওয়েটের সঙ্গে নানা রকম ক্যালকুলেশন করে আউটপুট দিত। এই নিউরাল নেটওয়ার্কে শুধু নতুন যুক্ত হয়েছে বায়াস। অর্থাৎ ওয়েট থাকে Edge -এ আর বায়াস থাকে Node -এ বা নিউরনে। যাই হোক, এই ওয়েট আর বায়াস-ই কিন্তু মুল ভ্যারিয়েবল, যেগুলোর মান এই মডেল অ্যাডজাস্ট করে নেয় ট্রেনিং করার সময়। আর এই দুটো ভ্যারিয়েবলকেই মডেল ভ্যারিয়েবল বলা হয়ে থাকে। এখন আমরা আমাদের মডেলের এই দুটো ভ্যারিয়েবলকে ডিফাইন করবো। এগুলো কিন্তু প্লেসহোল্ডার নয় যে এগুলোর মান বাইরে থেকে ইনপুট হবে। বরং এগুলো নর্মাল ভ্যারিয়েবল যেগুলো কিনা ট্রেনিং চলাকালীন অবস্থায় ক্যালকুলেশনের মধ্যে সময়ে সময়ে অ্যাডজাস্ট বা অপ্টিমাইজড হবে।

শুরুতেই আমরা ওয়েট ভ্যারিয়েবল ডিফাইন করি। একদম শুরুর উদাহরণে যেমন আমরা কিছু ওয়েট ডিফাইন করেছিলাম র‍্যান্ডোম ভ্যালু দিয়ে এবং তারপর ট্রেইন শুরু করেছিলাম। এবার আমরা সবগুলো ওয়েটের মান ধরব 0. ভয়ের কিছু নাই, ট্রেনিং শুরু হওয়া মাত্রই এগুলো বদলে ঠিক ঠাক ভ্যালুর দিকেই আগাবে। যা হোক, এদের সেইপ হবে [img_size_flat, num_classes]।

```
# Cell 13
weights = tf.Variable(tf.zeros([img_size_flat, num_classes]))
```

কেন এরকম হল? আবার সেই প্রথম উদাহরণের কথাই আনা যায় - ওখানে যেমন এক পাশে তিনটা ইনপুট নিউরন ছিল এবং আউটপুট লেয়ারে একটা নিউরন ছিল। আর আমাদের দরকার হয়েছিল 3x1 ওয়েট ম্যাট্রিক্স। ঠিক এই মডেলও যেহেতু লিনিয়ার মডেল (Accuracy নিয়ে মাথা ঘামাচ্ছি না আমরা, এবং বলছি না যে ক্লাসিফিকেশনের জন্য এটা ভালো কোন মডেল) আর এর প্রথম (ইনপুট) লেয়ারে 784 টা নিউরন আছে এবং আউটপুট লেয়ারে 10 টা নিউরন আছে তাই এর সেইপ এরকম। পরিষ্কার? :)

এ অবস্থায় আমার মনে হয় আমাদের মডেলটার একটা ভিজুয়ালাইজেশন দরকার। নিচে দিয়ে দিলাম,



এবার ডিফাইন করি bias এর জন্য ভ্যারিয়েবল। আগেও বলা হয়েছে, বায়াস থাকে নোডে বা নিউরনে, অর্থাৎ যখন একটি নিউরনের জন্য ওয়েট এবং এইজের ক্যালকুলেশন শেষ হয়ে নোডে জমা হয় তখন এর সাথে যোগ হয় bias. তো আমাদের ইনপুট লেয়ারে একগাদা নিউরন থাকলেও আউটপুট লেয়ারে কিন্তু ১০টাই নিউরন। তাই এই ডাটা অবজেক্ট (টেনসর বা ভেক্টর) এর সেইপ হবে [num_classes]। আর স্টেটমেন্ট হবে নিচের মত,

```
# Cell 14
biases = tf.Variable(tf.zeros([num_classes]))
```

মডেল

খুব সহজ ভাবে বলতে গেলে, আমাদের এই ম্যাথেম্যাটিক্যাল মডেলটি প্লেস হোল্ডার ভ্যারিয়েবল x এর মধ্যে থাকা ইমেজ গুলোর সাথে weights গুন করে এবং শেষে bias যোগ করে। আর এর রেজাল্টটি হয় একটি ম্যাট্রিক্স যার সেইপ হবে, [num_images, num_classes]. কারন কি? কারন হচ্ছে, ইনপুট লেয়ারের ডাটা ম্যাট্রিক্স বা x এর সেইপ হচ্ছে [num_images, img_size_flat] (None এর জায়গায় যেকোনো সংখ্যক ইমেজ দেয়া যাবে, মনে আছে? ধরলাম num_images সংখ্যক দিয়েছি), আর weights এর সেইপ হচ্ছে [img_size_flat, num_classes]. XxY আর YxZ গুন করলে XxZ -ই হয় :) তাই রেজাল্ট ম্যাট্রিক্সের সেইপ হবে, [num_images, num_classes]. আর এই ম্যাট্রিক্সের প্রত্যেকটি row এর সাথে bias ভেক্টর যোগ হয়। অর্থাৎ যদি ম্যাট্রিক্সে শুধু দুটো ইমেজের ডাটা থাকে সেগুলো হবে এরকম - [[c00, c01, c02, c03, c04, c05, c06, c07, c08, c09],[c10, c11, c12, c13, c14, c15, c16, c17, c18, c19]]. আর এগুলোর সাথে bias (দেখতে এমন - [b0, b1, b2, b3, b4, b5, b6, b7, b8, b9]) ভেক্টর যোগ করা যাবে খুব সহজে নিচের স্টেটমেন্ট দিয়ে।

```
# Cell 15
logits = tf.matmul(x, weights) + biases
```

জেনে রাখা ভালো - logits নামটি TensorFlow নিয়ে কাজ করার সময় একটা কনভেনশনাল নাম মাত্র। তো, logits হচ্ছে একটা ম্যাট্রিক্স যার row সংখ্যা = num_images এবং কলাম সংখ্যা = num_classes। আর তাই এভাবে বলা যায় - i তম row (ইমেজ) এর সাথে j তম কলাম (ক্লাস) -এর সম্পর্ক দেখেই ওই ইমেজের ক্লাস নির্ধারণ সম্ভব।

তবে সমস্যা হচ্ছে এখানে একটি row এর কলাম ভিত্তিক ভ্যালু গুলোর মান নানা রকম অর্থাৎ ছোট বড় মিলিয়ে। তাই কলামের ভ্যালু গুলো এই অবস্থায় সরাসরি দেখে ক্লাস বলা একটু টাফ। তাই আমরা যেটা করতে পারি, logits ম্যাট্রিক্সের প্রত্যেকটা row এর ভ্যালুগুলোকে নরমালাইজ করতে পারি যাতে করে পুরো একটা row এর ১০টি ভ্যালুর সমষ্টি ১ হয়। এবং প্রত্যেকটি ভ্যালু ০ থেকে ১ এর মধ্যে হয়। তাহলে এটাকে আমরা একটা প্রোব্যাবিলিটি ডিস্ট্রিউবিউশন বলতে পারি। যাই হোক, এই কাজটা করার জন্য আমরা softmax মেথডের সাহায্য নেবো নিচের মত,

```
# Cell 16
y_pred = tf.nn.softmax(logits)
```

তার মানে এবার প্রত্যেকটা row (প্রত্যেকটা ইমেজ) এর ১০ টা করে ভ্যালু (ক্লাস) হবে ০ থেকে ১ এর মধ্যে এবং যেটার ভ্যালু বেশি হবে অর্থাৎ প্রোব্যাব্লিলিটি সবচেয়ে বেশি হবে ধরা যায় ওই row (ইমেজ) এর ক্লাস/লেবেল ওইটা। ওইটা বলতে ১০টা কলামের ওই নাম্বার কলাম। তো, যে কলামের ভ্যালু বেশি সেই কলামের ইনডেক্স পেতে আমরা আগেও একবার ব্যবহার করেছি argmax মেথড। তাহলে আবার আমরা y_pred এর উপর এই মেথডের ইমপ্লিমেন্টেশন করে ফেলি।

```
# Cell 17
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

এখন পর্যন্ত আমরা কি করলাম? ইনপুট ইমেজ নেয়ার ব্যবস্থা করেছি। সেই ইনপুট ইমেজগুলোর টু ক্লাস ইনপুটের ব্যবস্থা করেছি। ইনপুট লেয়ার আর আউটপুট লেয়ার এর কাঠামো ঠিক করেছি। ইনপুট লেয়ার থেকে শূন্য ওয়েট গুন করে তারপর বায়াস যোগ করে আউটপুট লেয়ারে logits ম্যাট্রিক্স পেয়েছি এবং এটার উপর softmax, argmax আপ্লাই করে প্রত্যেকটি ইমেজের জন্য একটি করে ক্যালকুলেটেড ক্লাস পেয়েছি। অর্থাৎ ইনপুট নিয়ে মডেলের হিসাব করা রেজাল্ট পাওয়ার ব্যবস্থা হয়েছে। এরপর দেখতে হবে এই রেজাল্ট সঠিক ক্লাস থেকে কেমন ফারাকে আছে। তার উপর ভিত্তি করেই ওয়েট, বায়াস গুলো অ্যাডজাস্ট করবো।

Cost ফাংশন

মডেলকে এফিসিয়েন্ট করার জন্য আমাদেরকে weights, biases এগুলা ওপ্টিমাইজ করতে হবে। আর এগুলো অপ্টিমাইজ করার জন্য আমাদেরকে জানতে হবে মডেলটা কত ভালো পারফর্ম করল। এটা বের করতে পারবো y_pred এবং y_true এর তুলনা করে। এই তুলনা করার অনেক রকম পদ্ধতির মধ্যে একটি পদ্ধতি হচ্ছে cross-entropy মিজারমেন্ট। ক্লাসিফিকেশন সমস্যার জন্য এটি একটি বহুল ব্যবহৃত Cost Function. এটি একটি Continuous Function যার মান সবসময় পজিটিভ। কিন্তু যদি প্রেডিক্টেড আউটপুট এবং আমাদের চাহিদা মোতাবেক আউটপুট একদম মিলে যায় তাহলে এর মান আসে শূন্য। তার মানে, আমরা যদি এই ফ্যাক্টরটির দিকে নজর রাখি এবং সব সময় চাই যে, ওয়েট - বায়াস অ্যাডজাস্ট করে করে এই ফ্যাক্টরটির মান যতটা শূন্যের কাছাকাছি আনা যায় -তাহলেই আমরা ভালো Accuracy এর দিকে আগাবো। খুশির খবর হচ্ছে TensorFlow এর একটা বিল্ট ইন ফাংশন আছে এই ক্রস-এন্ট্রপি বের করার জন্য। মনে রাখতে হবে, এই ফাংশনটি logits ম্যাট্রিক্স নিয়ে কাজ করে। কারন এটি নিজেই softmax এর কাজটা ভিতরে করে ফেলে। তাই y_pred = tf.nn.softmax(logits) লাইনের আগের ভ্যালু আমাদের এখানে দরকার।

```
# Cell 18
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y_true)
```

যেহেতু আমরা একবারে পুরো logits ম্যাট্রিক্সকে ইনপুট হিসেবে দিয়েছি, তাই এই ম্যাট্রিক্সে যতগুলো ইমেজের ক্লাসিফিকেশন থাকবে সবগুলোর ক্রস-এন্ট্রপি বের করে ফেলছি উপরের লাইনে। তার মানে প্রত্যেকটি ইমেজের জন্যই আলাদা আলাদা ক্রস-এন্ট্রপি ভ্যালু পাবো আমরা। কিন্তু, পুরো সিস্টেমের অপটিমাইজেশন গাইড করার জন্য আমাদের একটা সিঙ্গেল ভ্যালু হলেই বরং ভালো। আর তাই, আমরা সাধারণ গড় করে নিতে পারি ওই আলাদা আলাদা ইমেজের আলাদা আলাদা ক্রস-এন্ট্রপি ভ্যালু গুলোর।

```
# Cell 19
cost = tf.reduce_mean(cross_entropy)
```

অপটিমাইজেশন মেথডঃ
তো, এখন যেহেতু আমাদের হাতে একটি Cost Measure থাকলো যেটাকে মিনিমাইজ করতে হবে, তাই এখন আমরা একটি অপ্টিমাইজার তৈরি করতে পারি। এই টিউটোরিয়ালে আমরা একটি ব্যাসিক অপ্টিমাইজার (Gradient Descent) ব্যবহার করবো। এটাও বিল্ট ইন আছে TensorFlow তে। আলাদা করে এটা বুঝতে এখানে ক্লিক করতে পারেন।

```
# Cell 20
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.5).minimize(cost)
```

এখানে 0.5 হচ্ছে গ্র্যাডিয়েন্ট খোজার স্টেপ সাইজ (যদি উপরের লিঙ্কে ক্লিক করে ঘুরে এসে থাকেন তাহলে এটা বুঝে ফেলার কথা)।

এখানে খুব গুরুত্বপূর্ণ একটা কথা মনে রাখা জরুরি - উপরের স্টেটমেন্ট এক্সিকিউট করার সাথে সাথেই কিন্তু অপটিমাইজেশন শুরু হয়ে যায় নি। TensorFlow গ্রাফে আমরা একটা নতুন অবজেক্ট যুক্ত করেছি মাত্র। আসলে কোন কিছুই হয়নি এখন পর্যন্ত। পুরো গ্রাফকে রান করালে এসব কাজ করা শুরু করবে একসাথে। যেহেতু আমরা জুপিটার নোটবুকে এই সেল/কোডব্লক গুলো এক্সিকিউট করছি তাই এগুলো রানটাইম মেমোরিতে থাকছে মাত্র।

পারফর্মেন্স মিজার করাঃ
ঘটনা যাই হোক, আমাদের একটা ফ্যাক্টর রেডি করা উচিৎ যার দিকে তাকিয়ে আমরা এক কথায় বলে দিতে পারবো আমাদের মডেলের পারফর্মেন্স কেমন। অর্থাৎ যদি মডেল তৈরি করে ট্রেইন করার শেষে আমরা একটা একটা করে টেস্ট ইমেজ ইনপুট দিয়ে আউটপুট জেনে খাতায় টালি করে করে হিসাব রাখতে চাই যে - কয়টা ভুল হল আর কয়টা ঠিক হল; তাহলে খবর আছে। বরং আমরা কুইক হিসাব করে বের করতে চাই আসলে ওভারঅল পারফর্মেন্স কেমন। যাতে করে আমরা মডেলের রিডিজাইন এবং মডিফিকেশনে দ্রুত মন দিতে পারি। ফাইনালি যখন মডেলটি পাবলিক ইউজের জন্য ছেড়ে দেয়া হবে, তখন সেখানে ইউজার একটা করে ইনপুট দিয়ে দেখবে কাজ করছে কিনা।

আমরা প্রথমে একটা বুলিয়ান ভেক্টর পেতে পারি নিচের মত করে,

```
# Cell 21
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

যার মধ্যে আছে প্রেডিক্টেড ক্লাস এবং টু ক্লাসের মিল চেক করে জমা করা কিছু টু, ফলস :) এরপর নিচের লাইনে আমরা সেই ভেক্টরটিকে আবার কাস্টিং করে ফ্লট এ কনভার্ট করছি যাতে করে টু মানে 1 এবং ফলস মানে 0 হয়। আর শেষ নাগাদ এগুলোর গড় বের করলেই কিন্তু বোঝা যাচ্ছে যে টোটাল Accuracy কেমন। তাই না? সব যদি টু হয় তাহলে ভেক্টরে থাকছে [1, 1, 1, 1, 1] (ধরি ৫টা ইমেজের ক্ষেত্রে) তাহলে গড় (যোগ দিয়ে ৫ দিয়ে ভাগ) তো 100% নাকি?

```
# Cell 22
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

TensorFlow রান

অনেক হয়েছে গ্রাফ সাজানো। এবার পুরো গ্রাফকে রান করার পালা আর এর ম্যাজিক দেখার পালা। তাহলে সেশন তৈরি করে ফেলিঃ

```
# Cell 23
session = tf.Session()
```

ভ্যারিয়েবল গুলোকে ইনিসিয়ালিয়াজ করে ফেলি,

```
# Cell 24
session.run(tf.global_variables_initializer())
```

কম রিসোর্সে অপটিমাইজেশন ফাংশনঃ
আমার জেনেছি যে - আমাদের ডাটাসেটে ৫০,০০০ ট্রেনিং ইমেজ আছে। যদি এই পুরো ডাটাসেটকে একবারেই আমাদের অপ্টিমাইজার ফাংশনের উপর দিয়ে দেই তাহলে কম্পিউটেশনে প্রচুর সময় লাগবে (যদি না GPU বা হাই পাওয়ার CPU বা মেশিন হয়) তাই আমরা যেটা করতে পারি, প্রত্যেকবার কিছু কিছু ইমেজ নিয়ে এর মধ্যে দিতে পারি। যেমন একবারে ১০০ করে দিতে পারি। এখানে ১০০ কে বলা হয় একটা batch.

```
# Cell 25
batch_size = 100
```

এই কাজ করার জন্য আমরা একটা হেল্পার ফাংশন তৈরি করে নেই নিচে,

```
# Cell 26
def optimize(num_iterations):
    for i in range(num_iterations):
        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = data.train.next_batch(batch_size)
        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph. # Note that the placehold
er for y_true_cls is not set # because it is not used during training.
        feed_dict_train = {x: x_batch,
                                           y_true: y_true_batch}
        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimizer.
        session.run(optimizer, feed_dict=feed_dict_train)
```

ভয়ের কি আছে? ফাংশনটা আসলে ৪ লাইনের :) শুধু খেয়াল করার বিষয় হচ্ছে feed_dict_train ভ্যারিয়েবলটা। এটার মাধ্যমেই কিন্তু প্লেসহোল্ডারের মধ্যে সত্যিকারের ইনপুট দেয়া হচ্ছে। এক্ষেত্রে x আর y_true কিন্তু আমরাই ডিক্লেয়ার করেছিলাম প্লেসহোল্ডার হিসেবে। আরেকটা হেল্পার ফাংশন আমারা বানিয়ে নিতে পারি পারফর্মেন্স শো করার জন্য. এটার ক্ষেত্রে ফিড ডিকশনারি হবে টেস্ট ইমেজ গুলো নিয়ে। নিচের মত,

```
# Cell 27
feed_dict_test = {x: data.test.images, y_true: data.test.labels,
                            y_true_cls: data.test.cls}
```

আর ফাংশনটি হবে,

```
# Cell 28
def print_accuracy():
    # Use TensorFlow to compute the accuracy.
    acc = session.run(accuracy, feed_dict=feed_dict_test)
    # Print the accuracy.
    print("Accuracy on test-set: {0:.1%}".format(acc))
```

ওয়েট ভিজুয়ালাইজ করাঃ
এই পার্টটি অপশনাল। যদি কেউ দেখতে চান ওয়েট ম্যাট্রিক্স গুলো দেখতে কেমন হচ্ছে তাহলে নিচের হেল্পার ফাংশন ব্যবহার করা যেতে পারে। ওয়েট ম্যাট্রিক্স যেহেতু একটা ম্যাট্রিক্স আর আমি ম্যাট্রিক্স মানেই মনে করি ইমেজ তাই এটাও দেখতে ইমেজের মত হবে। হোক না দুই কালার ওয়ালা।

```
# Cell 29
def plot_weights():
    # Get the values for the weights from the TensorFlow variable. w = session.run(wei
ghts)
    # Get the lowest and highest values for the weights. # This is used to correct the
 colour intensity across # the images so they can be compared with each other. w_min =
 np.min(w)
    w_max = np.max(w)
    # Create figure with 3x4 sub-plots,
    # where the last 2 sub-plots are unused. fig, axes = plt.subplots(3, 4) fig.subplo
ts_adjust(hspace=0.3, wspace=0.3)
    for i, ax in enumerate(axes.flat):
        # Only use the weights for the first 10 sub-plots.
        if i<10:
            # Get the weights for the i'th digit and reshape it. # Note that w.shape =
= (img_size_flat, 10)
            image = w[:, i].reshape(img_shape)
                            # Set the label for the sub-plot.
            ax.set_xlabel("Weights: {0}".format(i))
            # Plot the image.
            ax.imshow(image, vmin=w_min, vmax=w_max, cmap='seismic')
        # Remove ticks from each sub-plot.
        ax.set_xticks([])
        ax.set_yticks([])
```

> পরের টিউটোরিয়ালে কনভলিউশন বোঝার জন্য ওয়েট ম্যাট্রিক্স এর ভিজুয়ালাইজেশন বোঝা উপকারী।

কোন রকম অপ্টিমাজেশনের আগেই পারফর্মেন্স চেক করিঃ
আমরা যদি এই অবস্থায় নোটবুকের নতুন একটি সেলে নিচের কোড এক্সিকিউট করে accuracy দেখতে চাই,

```
# Cell 30
print_accuracy()
```

তাহলে আউটপুট আসবে,

```
Accuracy on test-set: 9.8%
```

কারন কি? মডেল ট্রেনিং করার আগেই কিভাবে শতকরা 10 ভাগ সঠিক উত্তর দেয়া শুরু করলো? আজব না? কারন হচ্ছে - আমাদের ওয়েট বায়াস শূন্য। তাই মডেল সব ইমেজকে প্রেডিক্ট করে শূন্য হিসেবে। কাকতালীয় ভাবে টেস্ট ইমেজ গুলোর মধ্যে শতকরা ১০ ভাগের মত ইমেজ ছিল শূন্যের। তাই সেগুলোর ক্ষেত্রে যখন প্রেডিক্টেড আর ট্রু ক্লাস মিলে গেছে, তাই accuracy আসতেছে 10% এর মত।

> ঝড়ে বক মরে ফকিরের কেরামতি বারে

অপটিমাইজেশন শুরু করিঃ
আমি এই পোস্টের শুরুর দিকের একটা উদাহরনেও একটা লুপকে ১টা সাইকেলে আটকে রেখে ভ্যালু গুলো নিয়ে যাচাই বাছাই এর কথা বলেছিলাম। এবারও সেরকম একটা এক্সপেরিমেন্ট করা যায়। আমরা একটি মাত্র অপটিমাইজেশন ইটারেশন করবো শুরুতে।

```
# Cell 31
optimize(num_iterations=1)
```

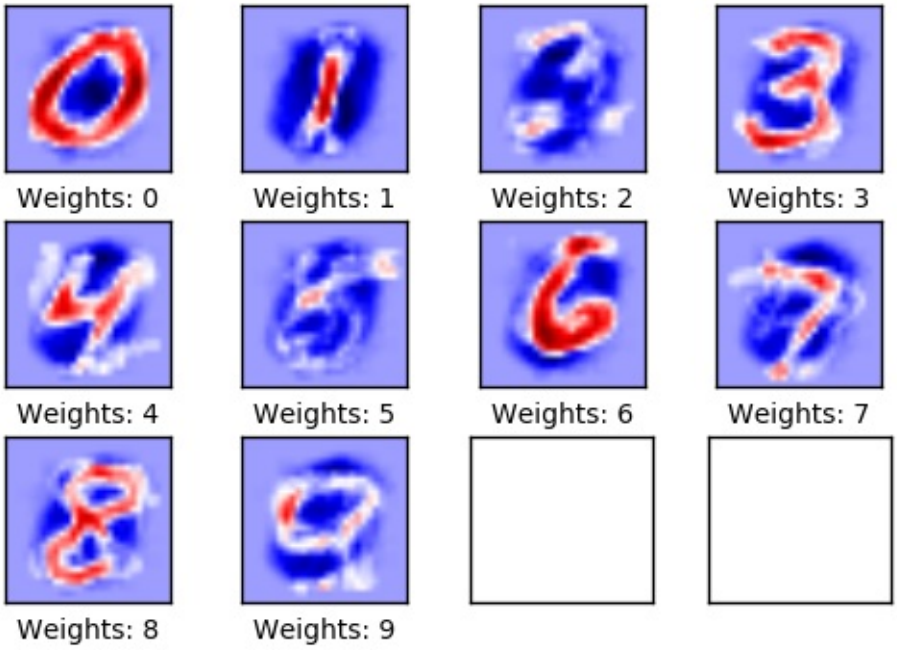
```
# Cell 32
print_accuracy()
```

```
Accuracy on test-set: 40.7%
```

একটা ইটারেশনেই প্রায় ৪০% সঠিক রেজাল্ট দিতে শিখেছে এই মডেল। বলে নেয়া ভালো - একটা ইটারেশনে কিন্তু এক ব্যাচ পরিমাণ ইমেজ নিয়ে কাজ করে মডেলটি। প্রত্যেক ইটারেশনে নতুন এবং পরবর্তী ব্যাচ (১০০টি) নিয়ে কাজ করে। optimize ফাংশনের কোড খেয়াল করুন। তো, এ অবস্থায় ওয়েট গুলো দেখতে চাইলে,

```
# Cell 33
plot_weights()
```

আউটপুট আসবে নিচের মত,



এখানে বলে নেয়া ভালো পজিটিভ ওয়েট গুলোকে লাল রং এবং নেগেটিভ ওয়েট গুলোকে নীল রং -এ প্রকাশ করা হয়েছে। মনে আছে অনেক আগের সেকশনে আমরা এরকম ওয়েট ম্যাট্রিক্স দিয়ে কনভলিউশনের ব্যাসিক ধারনা নিয়েছিলাম? যেখানে ম্যাট্রিক্স গুলো দেখতে ছিল [[+ -], [- +]] এরকম? এগুলোকেই ফিল্টার হিসাবে বলা হবে কনভলিউশন লেয়ারে। এখানে আমাদের মডেল ট্রেনিং করে শিখে এরকম ওয়েট ম্যাট্রিক্স ধরে নিয়েছে এবং দেখেছে যে এরকম ওয়েট ম্যাট্রিক্স হলে সেই রিলেটেড ফটো গুলোর সাথে রিয়েকশন পজিটিভ হয়। অর্থাৎ, যদি একটা 0 ওয়ালা ইমেজের সাথে এই ফিল্টারের দেখা হয় (এক্ষেত্রে ডাইরেক্ট x*W. কোন নির্দিষ্ট পার্টের সাথে কনভলিউশন নয়) তাহলে এই ফিল্টার সেই ফটোর সাথেই পজিটিভ রিয়েকশন করবে যার মধ্যে একটি সার্কেল টাইপ দাগ আছে। আর সেগুলোর সাথে নেগেটিভ রিয়েকশন করবে যেগুলোর মাঝখানটায় এক গাদা কালি আছে। তার মানে সে শূন্য লেখা আছে এমন ফটোর সাথে বেশি পজিটিভ ভ্যালু তৈরি করবে। এসব আরও বিস্তারিত বোঝা যাবে যাবে পরের টিউটোরিয়ালে যেখানে কনভলিউশনাল নিউরাল নেটওয়ার্ক ডিজাইন করা হবে এই মডেলকেই আরও ইফিসিয়েন্ট করার জন্য।

যা হোক, এবার ১০০০ অপটিমাইজেশন ইটারেশন করে দেখা যাকঃ

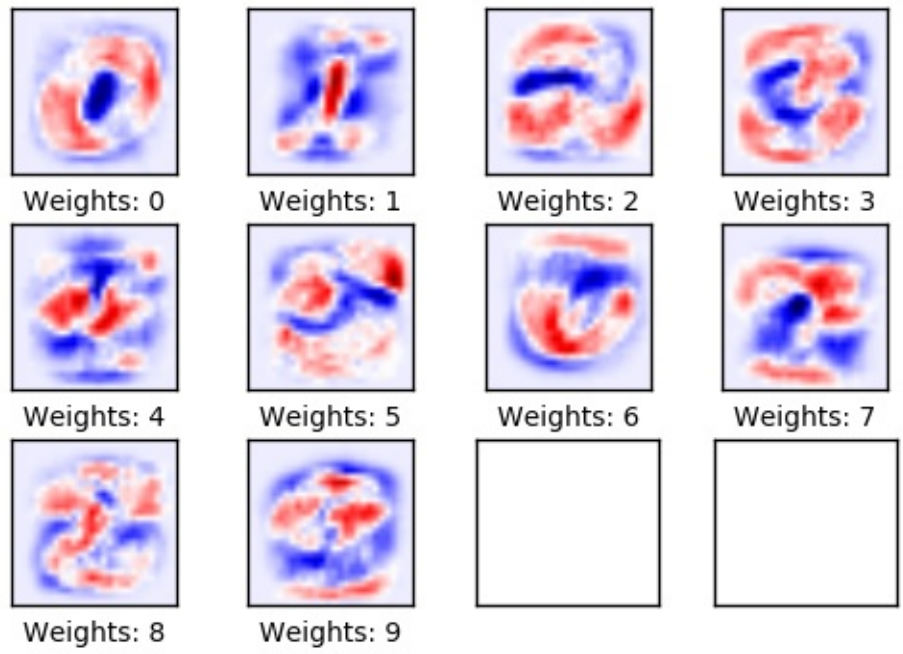
```
# Cell 34
# We have already performed 1 iteration already.
optimize(num_iterations=999)
```

```
# Cell 35
print_accuracy()
```

```
Accuracy on test-set: 91.7%
```

খেয়াল করুন, শুধুমাত্র লিনিয়ার মডেল ডিজাইন করেও ৯১% Accuracy পাওয়া গেছে। এটা সম্ভব হয়েছে ডিপ লার্নিং এর কারনেই। এখানে আমরা ইমেজ থেকে ফিচার এক্সট্র্যাক্ট করে দেই নি। শুধু ডাইরেক্ট পিক্সেল ভ্যালু গুলোকে ইনপুট লেয়ারে দিয়ে আউটপুট লেয়ারে টু ক্লাস দিয়ে ট্রেনিং করে মডেল ভ্যারিয়েবল গুলোকে অ্যাডজাস্ট করতে বলেছি। এতেই সে ওয়েট ম্যাট্রিক্স ধারনা করা শিখে গেছে। চাইলে এই অবস্থাতেও ওয়েট গুলো ভিজুয়ালাইজ করে দেখতে পারেন আর কোন প্রশ্ন থাকলে করতে পারেন।

```
# Cell 36
plot_weights()
```



এখানে দেখা যাচ্ছে ওয়েট ম্যাট্রিক্স গুলো আরও একটু জটিল হিসাবে মগ্ন। অর্থাৎ এমন না যে সার্কেল ধরে পজিটিভ ওয়েট সেট করেছে (যেমন 0 এর ক্ষেত্রে)। বরং একটু ছাড়া ছাড়া ভাবে। এটা সে করতে বাধ্য হয়েছে এক এক জনের এক এক রকম শূন্য লেখার সঙ্গে নিজেকে মানিয়ে নিতে গিয়ে।

অনেক হল গবেষণা। তো, এবার আমরা TensorFlow এর সেশন ক্লোজ করতে পারি নিচের মত।

```
# Cell 37
session.close()
```

> উপরের আলোচ্য ধাপ গুলো নিয়ে পূর্ণ .ipynb ডকুমেন্টটি পাওয়া যাবে এখানে

পূর্ণ প্রোগ্রামঃ যারা নোটবুকে ধাপে ধাপে এই কোড ব্লক গুলো এক্সিকিউট করেছেন বোঝার জন্য এবং এখন গোছানো একটা প্রোগ্রাম চান যেকোনো জায়গায় রান করার জন্য - ক্লিক করুন এখানে।