ECE598: Representation Learning: Algorithms and Models Fall 2017

Lecture 9: Neural Networks: Architecture of Neural Networks

Lecturer: Prof. Pramod Viswanath *Scribe: Moitreya Chatterjee, Sep 26, 2017*

Sitting on your shoulders is the most complicated object in the known universe - Michio Kaku.

9.1 Discriminative Models

In the context of probability and statistics, *Discriminative Models* are approaches where the dependence of the unobserved variables conditioned on the observed ones is modeled directly.

Note: This is in contrast to *Generative Models*, which are a class of approaches that model all random variables associated with a phenomenon, both those that can be observed as well as the unobserved ones [Bis06].

The contrast is exemplified by the following classification task. Let's assume that we are given data, $\mathcal{X}$, i.e. the observed variable and we want to determine its class label, $\mathcal{Y}$, the unobserved (target) variable. A discriminative classifier, such as a Logistic Regression, would directly model the posterior class probabilities, i.e. $\mathrm{P}(\mathcal{Y}|\mathcal{X})$, to perform this inference. Alternatively, some other discriminative models also map out a deterministic relation of $\mathcal{Y}$ as a function of $\mathcal{X}$, *Artificial Neural Networks* being a prime example.

A generative classifier, such as Naive Bayes, on the other hand, makes use of the joint distribution of $\mathcal{X}$ and $\mathcal{Y}$, i.e. $\mathrm{P}(\mathcal{X},\mathcal{Y})$ to perform this inference.

Kaku's quote cited above embodies the power of discriminative models elegantly, since the human brain houses a complicated network of neurons that are known to essentially perform a discriminative reasoning, almost effortlessly all the time.

9.2 Perceptron

The history of Artificial Neural Networks, commences with the seminal work of Rosenblatt in 1958 [Ros58]. In this work, he first proposed a structure and described the functioning of an artificial neuron, inspired from their counterparts found in the brains of mammals.

Figure 9.1 shows a representative schematic diagram of a Perceptron, as conceived by Rosenblatt.

9.2.1 Description of the Architecture

The Perceptron consists of an n-dimensional input, $\vec{x} \in \mathbb{R}^n$, amplified by a corresponding input weight vector $\vec{w} \in \mathbb{R}^n$, a scalar output variable $y \in \mathbb{R}$, and an input bias of the cell, $w_0 \in \mathbb{R}$. The final output of the cell is related to the input by the following:

$$y = \sigma(\vec{w}^T\vec{x} + w_0),$$

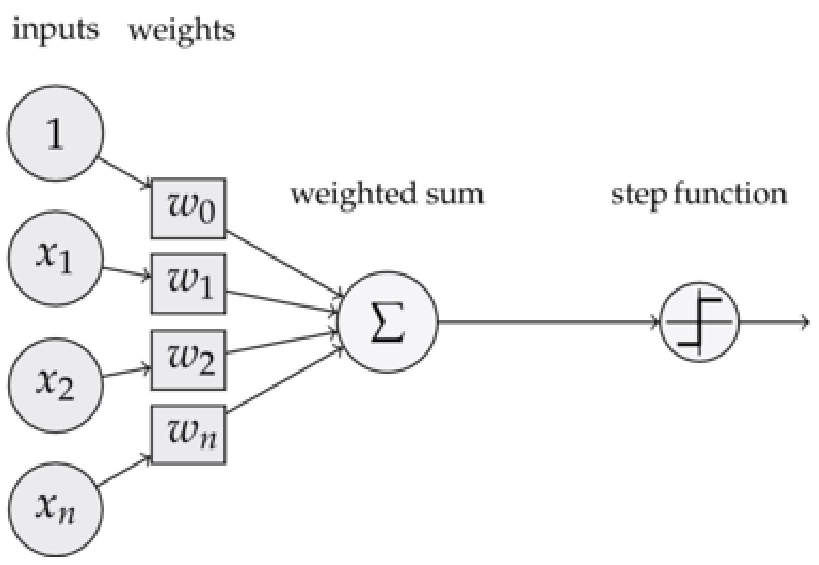


Figure 9.1: A schematic representation of a Perceptron.

where sigma is an *Activation Function* and is typically non-linear. In the context of Figure 9.1, the step function has been chosen as the activation function.

Note: It is interesting to note here that the weighted combination of inputs as represented in a perceptron, is simply a linear model. The only opportunity of making this model sufficiently complex is by injecting non-linearity via the choice of appropriate activation function.

Note: Further, while modeling a process with neural networks, we are not restricted to use just one perceptron but can use a stack of them in different configurations, as well. This setting is what is typically known as *Neural Networks*.

9.2.2 Activation Functions

Activation Functions are functions, which map the weighted sum of inputs to a neuron, to its output. The idea behind introducing them in the Perceptron, was to have a function that behaved a lot like the *Action Potential* of the neuron cell [Ros58].

Activation Functions typically allow for the introduction of non-linearity in neural networks. Typically, the following are some of the desirable properties of activation functions [GBC16]:

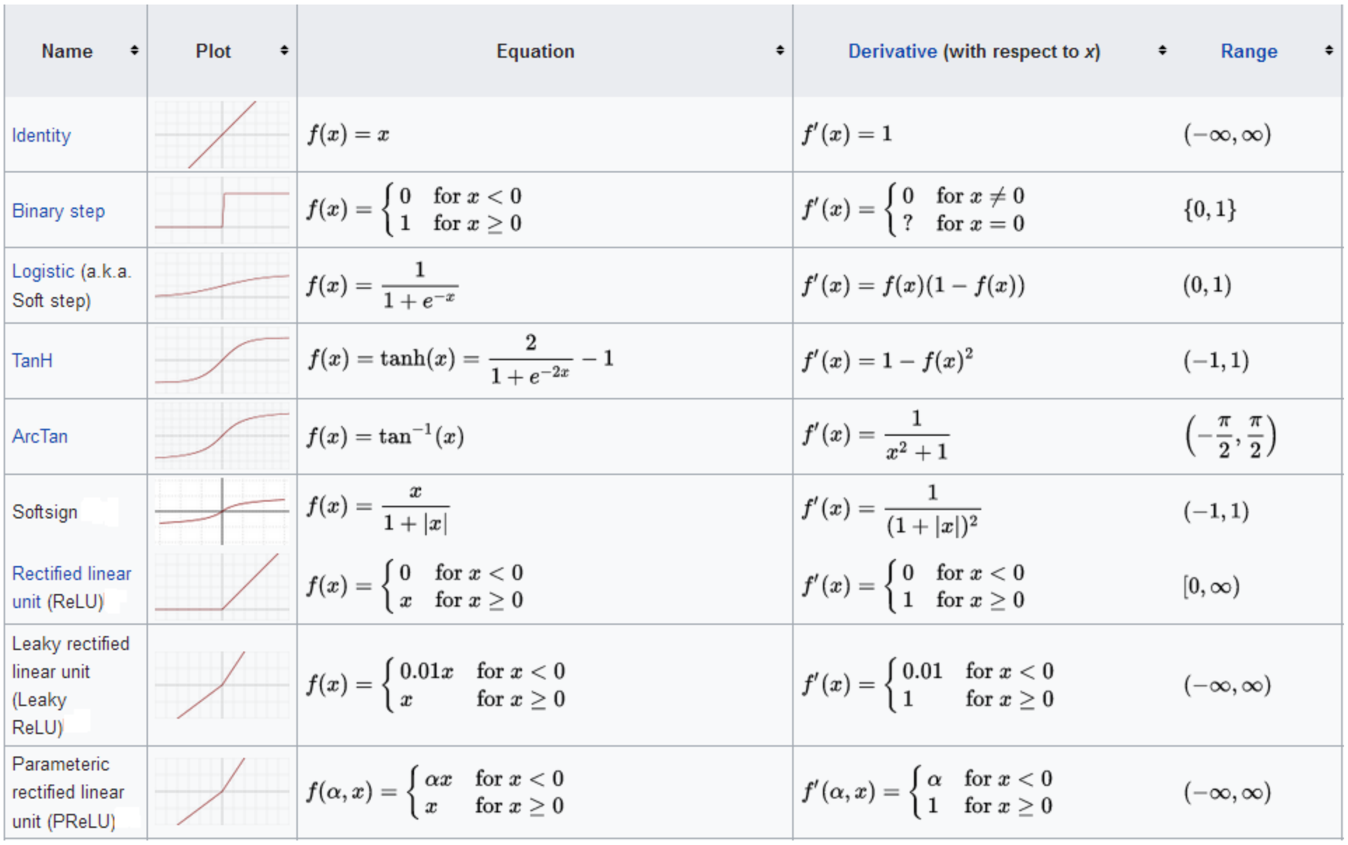
- *Non-Linearity*: When the activation function is non-linear, then a two-layer neural network can be proven to be a universal function approximator [Cyb89]. On the other hand, linear activation functions result in essentially the neural network becoming a linear model, which often wounds up being suboptimal for various tasks.
- *Continuous Differentiability*: This property is necessary for enabling gradient-based optimization methods, especially during training.
- *Range of the Function*: The range of the activation function plays a crucial role in efficient training of the neural network. Bounded range typically results in training that converges

quickly, while unbounded ones make the task of weight tuning harder.

- *Monotonicity*: When the activation function is monotonic, the error surface associated with a single-layer model is guaranteed to be convex. This makes the task of obtaining optimal neuron weights, significantly easier.
- *Approximates the Identity Near the Origin*: Such activation functions allow the neural network to efficiently learn when its weights are initialized with small random values.

Choice of Activation Functions

A wide assortment of activation functions have been explored by the community so far by the neural networks community.



Name	Plot	Equation	Derivative (with respect to *x*)	Range
Identity		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$	$\{0, 1\}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$	$\left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$
Softsign		$f(x) = \frac{x}{1+\|x\|}$	$f'(x) = \frac{1}{(1+\|x\|)^2}$	$(-1, 1)$
Rectified linear unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$
Leaky rectified linear unit (Leaky ReLU)		$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Parameteric rectified linear unit (PReLU)		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$

Figure 9.2: A table showing some common activation functions, their gradient with respect to the input, and their range [ali].

Figure 9.2 shows a list of some popular activation functions, their response plots, the equations which govern their response, their corresponding derivatives with respect to their inputs, and the range of their outputs. While not all of the functions possess the "desirable" properties listed above, but they still tend to behave quite robustly, while training. For instance, the Rectified Linear Unit (ReLU) activation is not fully differentiable but it is still compatible with gradient based training methods because it is not differentiable only at one point in the input space. Further, it is also noteworthy that the gradients of many of these functions, can be expressed in terms of the output of the original function. This is true for example for the Logistic, and the tan-hyperbolic activations. This makes the task of computing the derivatives much easier.

9.2.3 Artificial Neural Networks

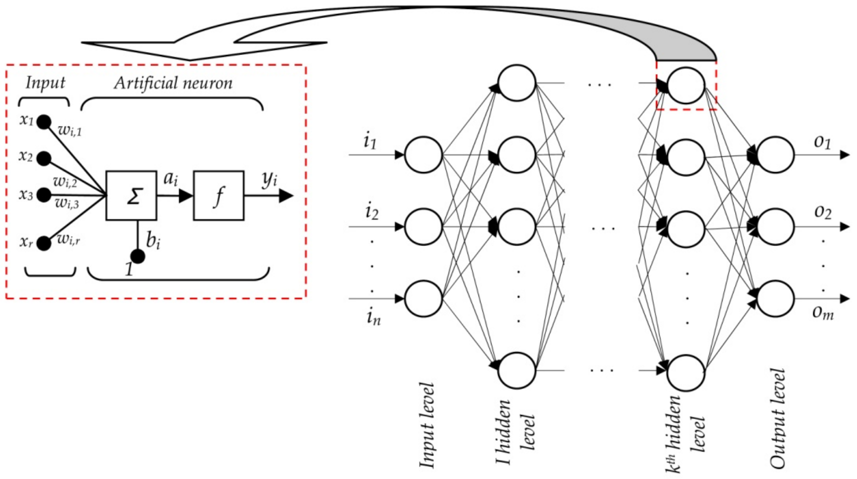


Figure 9.3: A schematic representation of an Artificial Neural Network.

So far in our discussion, we have only looked at a single perceptron cell. However, in a landmark work by Fukushima et al. [FMI83], the idea of stacking up perceptrons to build a "neural network" was proposed. Such networks were capable of approximating more complicated functions, such as those arising in tasks such as in handwritten numeral recognition. Figure 9.3 shows a representative example of an artificial neural network.

In his pioneering work Cybenko, observed that with a non-linear activation function, a two-layer neural network can be proven to be a universal function approximator [Cyb89]. This possibly helps explain, atleast partially, their ubiquity in applications relevant to today's day and age, while at the same time giving us a sense of why they have such amazing generalizability.

9.2.4 Deep Neural Networks

Given the capacity to approximate extremely complex functions, the neural network community, in order to apply neural nets to different applications, started designing neural network architectures that were several layers deep. Such networks are, intuitively, called *Deep Neural Networks*. Going back to our definition of parametric discriminative models, we can represent a deep neural network as a *Composite Functional Mapping* from the input, say $\vec{x}$ to the output, say y. This is given by:

$$y = f_1(\vec{w_1}, f_2(\vec{w_2}, ..., f_n(\vec{w_n}, \vec{x})))$$

Over the years, with increasing insight about such networks the community has calibrated different forms of these functions, f_i's.

Depending on whether the network accepts *fixed length* inputs or *variable length* ones, we pick the functions appropriately. In the following section, we will look at the fixed-length setting, in the context of image inputs, while later on we will delve into variable-length inputs such as text. Yet other forms of Neural Network architectures exists and the interested reader is referred to [zoo].

9.3 Convolutional Neural Networks

In the late 80's *Convolutional Neural Networks* (CNNs) were introduced for the task of image classification [LBD+90], specifically in the context of recognizing hand-written digits. Such networks became immensely popular in the recent times, owing to their high precision in recognizing image content. Modern day, CNNs tend to be deeper and typically take the form of the network, shown in Figure 9.4.

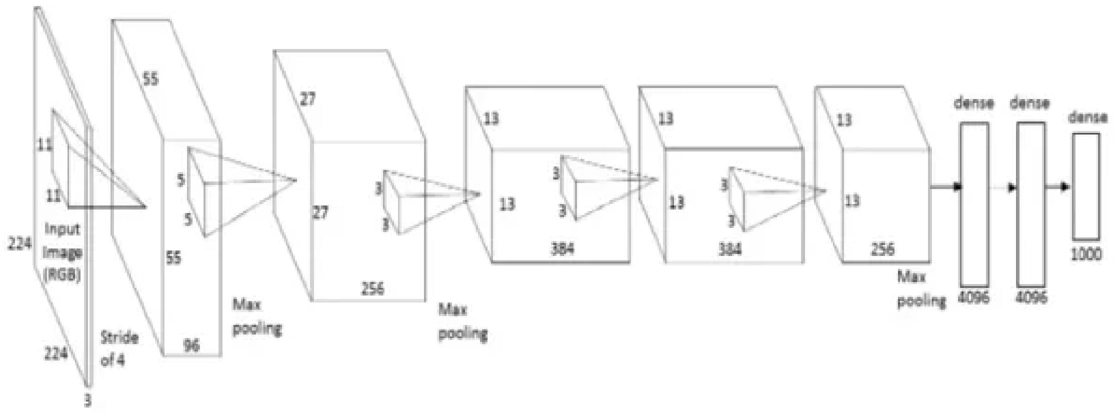


Figure 9.4: A block diagram representing the AlexNet Network.

Such networks typically have the following components:

- *Convolution Layers*: One key property of images is that they are *Shift Invariant Signals*, i.e. an image of a dog should be categorized as dog no matter where the dog is positioned in the image. This property motivated the design of convolution layers, where a weight matrix, typically much smaller than the image is convolved with the image and its response is noted. Mathematically, these layers perform a set of convolution operations, represented by:

  $$a^l = a^{l-1} * W^l + b^l,$$

  where a^i denotes the activation from layer i, W^i, b^i denote the convolution weight kernel and the bias of that kernel at layer i, and * indicates convolution.

  Note: The above convolution may also be represented with a simple matrix product, but then W must be a *Toeplitz* Matrix, so that it ensures shift invariance.

- *Maxpool Layers*: These layers serve the purpose of downsampling the response map. A typical Maxpool operation selects only the maximum response in a window of size, say 3×3. The motivation behind performing a maxpool is that only the maximum response is retained while the response at everywhere else is considered local variations in the data, i.e. it is not considered key to the final decision making.

- *Fully Connected Layers*: Typically, after several stages of Convolution followed by Maxpool, one ends up using a set of Fully Connected Layers in a CNN. These layers resemble the familiar densely connected Neural Networks shown in Figure 9.3. These layers perform an operation that may be represented as a simple matrix multiply, like the one shown below:

$$a^l = a^{l-1}W^l,$$

where the notations carry forward their meaning from before.

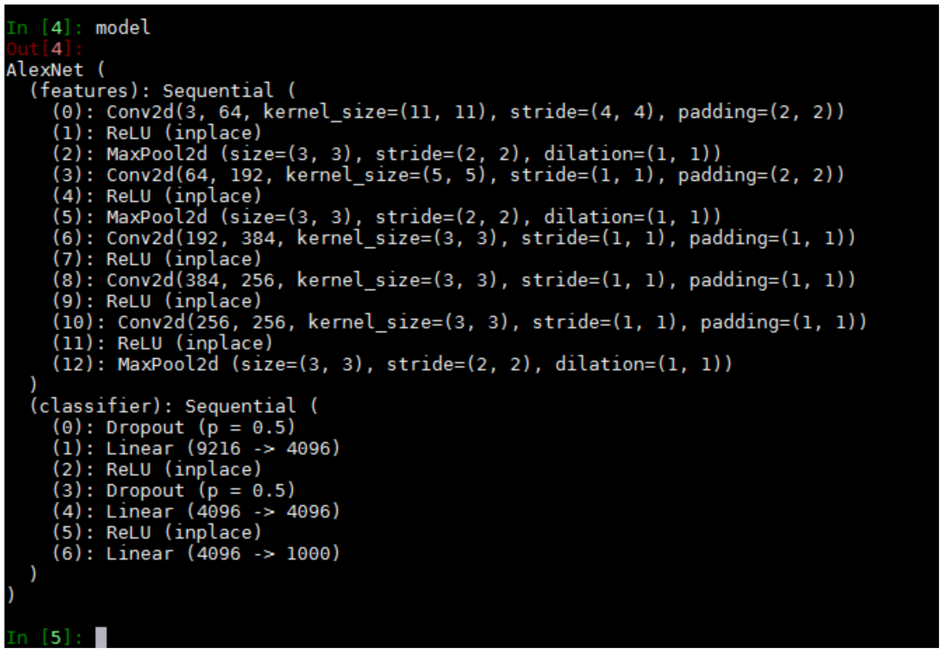
- *Softmax Layers*: These layers convert the final output from the CNN to probability scores, indicating how likely it is that the sample belongs to a certain class. This is given by:

$$P(y_l) = \frac{\exp(h_l)}{\sum_{l'} \exp(h_{l'})},$$

where h_i is the response of the i^{th} neuron in the final layer. Typically, the h_i's are obtained via a weighted linear combination of the responses from the previous layer. This essentially amounts to performing a multi-class logistic regression, which is essentially a linear classifier. Therefore, to increase the classification capacity of the network, some recent work consider replacing this with non-parametric classifiers, such as k-Nearest Neighbor [knn]. Nonetheless softmax remains popular as a simple classifier, which permits gradient-based backpropagation.

Once the softmax probabilites are obtained, the image is typically assigned the class with the highest probability score.

Note: Each of the above listed operations can be thought of as functions mapping the input to the output class labels.

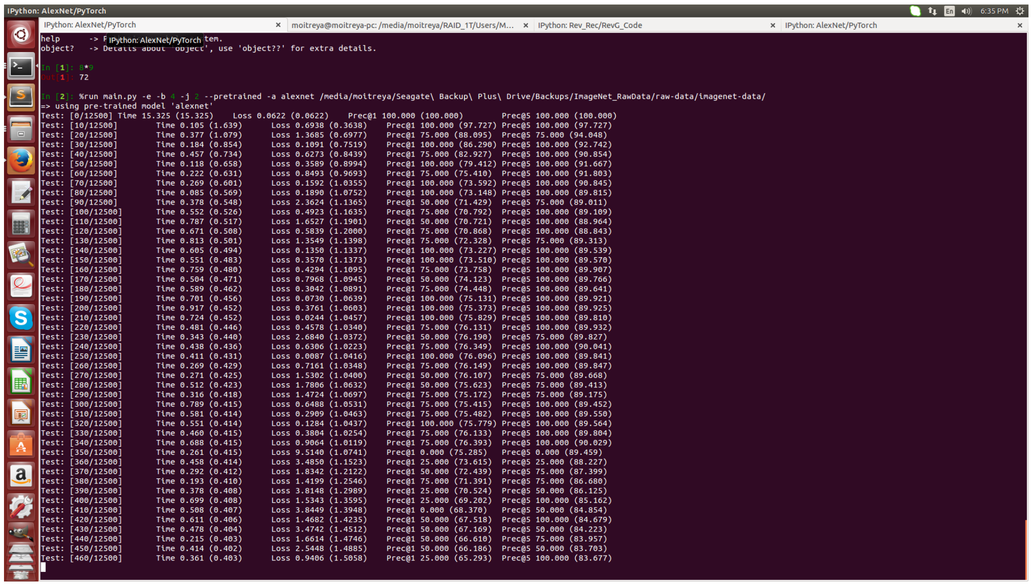


```
In [4]: model
Out[4]:
AlexNet (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU (inplace)
    (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU (inplace)
    (5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU (inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU (inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU (inplace)
    (12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  )
  (classifier): Sequential (
    (0): Dropout (p = 0.5)
    (1): Linear (9216 -> 4096)
    (2): ReLU (inplace)
    (3): Dropout (p = 0.5)
    (4): Linear (4096 -> 4096)
    (5): ReLU (inplace)
    (6): Linear (4096 -> 1000)
  )
)

In [5]:
```

Figure 9.5: A class definition of AlexNet in PyTorch.

Figure 9.6 shows an execution of the AlexNet architecture [KSH12], represented in Figure 9.4 on the Validation Set of the ImageNet dataset, while Figure 9.5 shows the definition of the CNN in the popular deep learning library, PyTorch.



```
IPython: AlexNet/PyTorch
IPython: AlexNet/PyTorch   moitreya@moitreya-pc: /media/moitreya/RAID_1T/Users/M...   IPython: Rev_Rec/RevG_Code   IPython: AlexNet/PyTorch
help      -> F IPython: AlexNet/PyTorch tem.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: 8*9
Out[1]: 72

In [2]: %run main.py -e -b 4 -j 2 --pretrained -a alexnet /media/moitreya/Seagate\ Backup\ Plus\ Drive/Backups/ImageNet_RawData/raw-data/imagenet-data/
=> using pre-trained model 'alexnet'
Test: [0/12500] Time 15.325 (15.325)    Loss 0.0622 (0.0622)    Prec@1 100.000 (100.000)        Prec@5 100.000 (100.000)
Test: [10/12500]        Time 0.105 (1.639)      Loss 0.6938 (0.3638)    Prec@1 100.000 (97.727) Prec@5 100.000 (97.727)
Test: [20/12500]        Time 0.377 (1.079)      Loss 1.3685 (0.6977)    Prec@1 75.000 (88.095)  Prec@5 75.000 (94.048)
Test: [30/12500]        Time 0.184 (0.854)      Loss 0.1091 (0.7519)    Prec@1 100.000 (86.290) Prec@5 100.000 (92.742)
Test: [40/12500]        Time 0.457 (0.734)      Loss 0.6273 (0.8439)    Prec@1 75.000 (82.927)  Prec@5 100.000 (90.854)
Test: [50/12500]        Time 0.118 (0.658)      Loss 0.3589 (0.8994)    Prec@1 100.000 (79.412) Prec@5 100.000 (91.667)
Test: [60/12500]        Time 0.222 (0.631)      Loss 0.8493 (0.9693)    Prec@1 75.000 (75.410)  Prec@5 100.000 (91.803)
Test: [70/12500]        Time 0.269 (0.601)      Loss 0.1592 (1.0355)    Prec@1 100.000 (73.592) Prec@5 100.000 (90.845)
Test: [80/12500]        Time 0.085 (0.569)      Loss 0.1890 (1.0752)    Prec@1 100.000 (73.148) Prec@5 100.000 (89.815)
Test: [90/12500]        Time 0.378 (0.548)      Loss 2.3624 (1.1365)    Prec@1 50.000 (71.429)  Prec@5 75.000 (89.011)
Test: [100/12500]       Time 0.552 (0.526)      Loss 0.4923 (1.1635)    Prec@1 75.000 (70.792)  Prec@5 100.000 (89.109)
Test: [110/12500]       Time 0.787 (0.517)      Loss 1.6527 (1.1901)    Prec@1 50.000 (70.721)  Prec@5 100.000 (88.964)
Test: [120/12500]       Time 0.671 (0.508)      Loss 0.5839 (1.2000)    Prec@1 75.000 (70.868)  Prec@5 100.000 (88.843)
Test: [130/12500]       Time 0.813 (0.501)      Loss 1.3549 (1.1398)    Prec@1 75.000 (72.328)  Prec@5 75.000 (89.313)
Test: [140/12500]       Time 0.605 (0.494)      Loss 0.1350 (1.1337)    Prec@1 100.000 (73.227) Prec@5 100.000 (89.539)
Test: [150/12500]       Time 0.551 (0.483)      Loss 0.3570 (1.1373)    Prec@1 100.000 (73.510) Prec@5 100.000 (89.570)
Test: [160/12500]       Time 0.759 (0.480)      Loss 0.4294 (1.1095)    Prec@1 75.000 (73.758)  Prec@5 100.000 (89.907)
Test: [170/12500]       Time 0.504 (0.471)      Loss 0.7968 (1.0945)    Prec@1 50.000 (74.123)  Prec@5 100.000 (89.766)
Test: [180/12500]       Time 0.589 (0.462)      Loss 0.3042 (1.0891)    Prec@1 75.000 (74.448)  Prec@5 100.000 (89.641)
Test: [190/12500]       Time 0.701 (0.456)      Loss 0.0730 (1.0639)    Prec@1 100.000 (75.131) Prec@5 100.000 (89.921)
Test: [200/12500]       Time 0.917 (0.452)      Loss 0.3761 (1.0603)    Prec@1 100.000 (75.373) Prec@5 100.000 (89.925)
Test: [210/12500]       Time 0.724 (0.452)      Loss 0.0244 (1.0457)    Prec@1 100.000 (75.829) Prec@5 100.000 (89.810)
Test: [220/12500]       Time 0.481 (0.446)      Loss 0.4578 (1.0340)    Prec@1 75.000 (76.131)  Prec@5 100.000 (89.932)
Test: [230/12500]       Time 0.343 (0.440)      Loss 2.6840 (1.0372)    Prec@1 50.000 (76.190)  Prec@5 75.000 (89.827)
Test: [240/12500]       Time 0.438 (0.436)      Loss 0.6306 (1.0223)    Prec@1 75.000 (76.349)  Prec@5 100.000 (90.041)
Test: [250/12500]       Time 0.411 (0.431)      Loss 0.0087 (1.0416)    Prec@1 100.000 (76.096) Prec@5 100.000 (89.841)
Test: [260/12500]       Time 0.269 (0.429)      Loss 0.7161 (1.0348)    Prec@1 75.000 (76.149)  Prec@5 100.000 (89.847)
Test: [270/12500]       Time 0.271 (0.425)      Loss 1.5302 (1.0400)    Prec@1 50.000 (76.107)  Prec@5 75.000 (89.668)
Test: [280/12500]       Time 0.512 (0.423)      Loss 1.7806 (1.0632)    Prec@1 50.000 (75.623)  Prec@5 75.000 (89.413)
Test: [290/12500]       Time 0.316 (0.418)      Loss 1.4724 (1.0697)    Prec@1 75.000 (75.172)  Prec@5 75.000 (89.175)
Test: [300/12500]       Time 0.789 (0.415)      Loss 0.6488 (1.0531)    Prec@1 75.000 (75.415)  Prec@5 100.000 (89.452)
Test: [310/12500]       Time 0.581 (0.414)      Loss 0.2909 (1.0463)    Prec@1 75.000 (75.482)  Prec@5 100.000 (89.550)
Test: [320/12500]       Time 0.551 (0.414)      Loss 0.1284 (1.0437)    Prec@1 100.000 (75.779) Prec@5 100.000 (89.564)
Test: [330/12500]       Time 0.460 (0.415)      Loss 0.3804 (1.0254)    Prec@1 75.000 (76.133)  Prec@5 100.000 (89.804)
Test: [340/12500]       Time 0.688 (0.415)      Loss 0.9064 (1.0119)    Prec@1 75.000 (76.393)  Prec@5 100.000 (90.029)
Test: [350/12500]       Time 0.261 (0.415)      Loss 9.5140 (1.0741)    Prec@1 0.000 (75.285)   Prec@5 0.000 (89.459)
Test: [360/12500]       Time 0.458 (0.414)      Loss 3.4850 (1.1523)    Prec@1 25.000 (73.615)  Prec@5 25.000 (88.227)
Test: [370/12500]       Time 0.292 (0.412)      Loss 1.8342 (1.2122)    Prec@1 50.000 (72.439)  Prec@5 75.000 (87.399)
Test: [380/12500]       Time 0.193 (0.410)      Loss 1.4199 (1.2546)    Prec@1 75.000 (71.391)  Prec@5 75.000 (86.680)
Test: [390/12500]       Time 0.378 (0.408)      Loss 3.8148 (1.2989)    Prec@1 25.000 (70.524)  Prec@5 50.000 (86.125)
Test: [400/12500]       Time 0.699 (0.408)      Loss 1.5343 (1.3595)    Prec@1 25.000 (69.202)  Prec@5 100.000 (85.162)
Test: [410/12500]       Time 0.508 (0.407)      Loss 3.8449 (1.3948)    Prec@1 0.000 (68.370)   Prec@5 50.000 (84.854)
Test: [420/12500]       Time 0.611 (0.406)      Loss 1.4682 (1.4235)    Prec@1 50.000 (67.518)  Prec@5 100.000 (84.679)
Test: [430/12500]       Time 0.478 (0.404)      Loss 3.4742 (1.4512)    Prec@1 50.000 (67.169)  Prec@5 50.000 (84.223)
Test: [440/12500]       Time 0.215 (0.403)      Loss 1.6614 (1.4746)    Prec@1 50.000 (66.610)  Prec@5 75.000 (83.957)
Test: [450/12500]       Time 0.414 (0.402)      Loss 2.5448 (1.4885)    Prec@1 50.000 (66.186)  Prec@5 50.000 (83.703)
Test: [460/12500]       Time 0.361 (0.403)      Loss 0.9406 (1.5058)    Prec@1 25.000 (65.293)  Prec@5 100.000 (83.677)
```

Figure 9.6: An execution of AlexNet in PyTorch.

9.4 Types of CNN Architectures

The quest for models with improved performances at image classification tasks has driven the community to design newer CNN architectures, beyond just AlexNet. Here, we enlist some of the other popular, successful architectures for image classification and briefly discuss their characteristics.

- *VGG*: VGG is a suite of popular CNN architectures, proposed by the Visual Geometry Group at Oxford University, originally designed for classifying ImageNet images [SZ14]. This class of CNNs, majorly adopt the AlexNet architecture, except that these networks are deeper. The VGG architectures are either 11, 13, 16, or 19 layers deep. The 19 layer architecture had a top-5 misclassification rate of 8.0%.

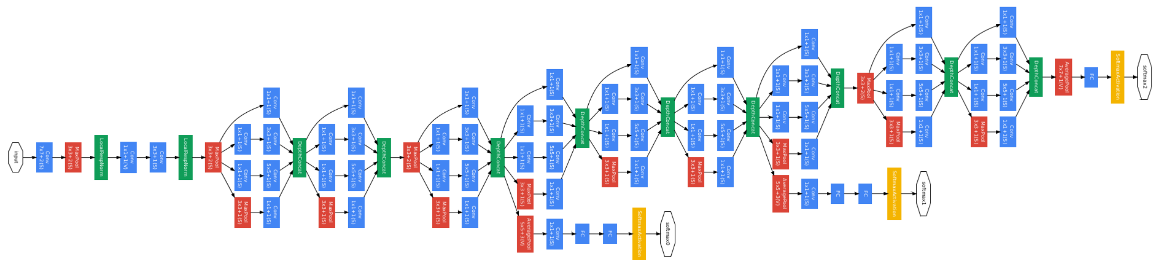


Figure 9.7: A schematic representation of the architecture of GoogleNet.

- *GoogleNet*: GoogleNet is another successful CNN architectures, proposed by Google, for classifying ImageNet images [SLJ+15]. This CNN had 22 layers. However, one of the interesting insights from their work was that training deeper networks isn't always trivial

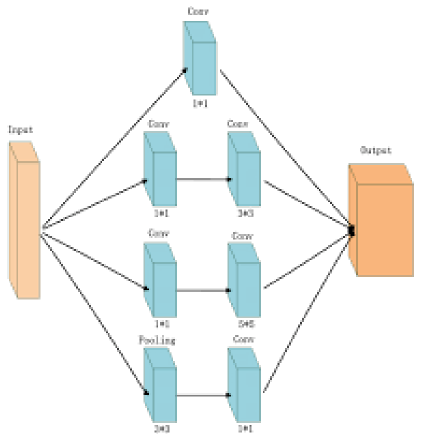


Figure 9.8: A schematic representation of the Inception Module in GoogleNet.

and one often encounters challenges related to vanishing gradients. They thus introduced Softmax layers from intermediate levels, so that more supervisory signal could be introduced. Figure 9.7 shows a schematic representation of the architecture of GoogleNet. The other revelation of this work was that performing 1×1 convolution in "Inception" module structure helped introduce greater non-linearity. Figure 9.8 shows a schematic diagram of an inception module. This architecture had a top-5 misclassification rate of about 7.0%.

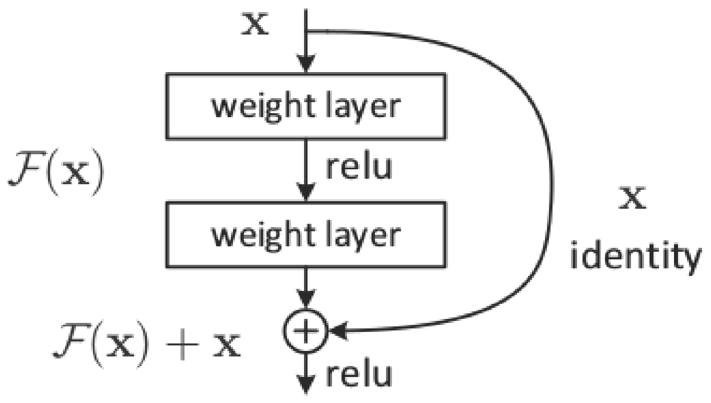


Figure 9.9: A schematic representation of the Skip Connections in ResNet.

- *ResNet*: ResNet, short for Residual Network, is one of the most successful CNN architectures, proposed by Microsoft Research, for classifying ImageNet images [HZRS16]. This CNN had 152 layers. The authors of this work too emphasized that training deeper networks isn't always trivial and challenges related to vanishing gradients and overfitting are common. They thus introduced Skip Connections in the intermediate layers, so that the signal could potentially skip a layer and move over to a subsequent layer, if necessary. This also holds for the gradient during back-propagation. Figure 9.9 shows a schematic representation of skip connections that are pervasive in the entire network. With this architecture the top-5 misclassification rate on ImageNet went down to about 3.5%, which exceeds human performance as well.

Note: To obtain increasingly better performance, one typically needed to design deeper architectures.

9.5 Applications of CNNs

CNNs have been applied for a wide variety of tasks, especially those in computer vision, such as detecting objects in an image [RDGF16], matching images [AJM15], style transfer between images [GEB16], and so on but also in natural language processing, such as for modeling sentence-pairs [YSXZ15].

9.6 Recurrent Neural Networks

While CNNs can effectively model fixed-length inputs, variable-length inputs, such as those arising in text, or in temporal data, require different deep neural network architectures. A popular class of deep neural network architectures that have been successful at modeling these kind of inputs is known as *Recurrent Neural Networks* (RNNs). These set of architectures were inspired by Hopfield Networks [Hop82].

Principally, there are 3 main varieties of RNNs, viz.

- Vanilla Recurrent Neural Networks (Vanilla RNN)
- Long-Short Term Memory Networks (LSTMs)
- Gated Recurrent Units (GRUs)

We will look into all 3 of them, in some level of detail.

9.7 Vanilla RNN

A *Vanilla Recurrent Neural Network* (Vanilla RNN) is a neural network that consists of a pair of input and output and a hidden-state. For modeling data, these units are essentially replicated over time. The output at any given time-step is dependent on the current input, and the previous

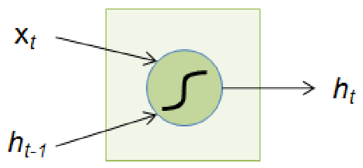


Figure 9.10: A schematic representation of the Vanilla RNN Cell.

hidden state. This hidden state, essentially, acts as a memory of the network. Figure 9.10 shows the architecture of a RNN cell.

Vanilla RNNs can be thought of as duplications of the RNN cells over time, as shown in Figure 9.11, where the cells share the weights across time-steps.

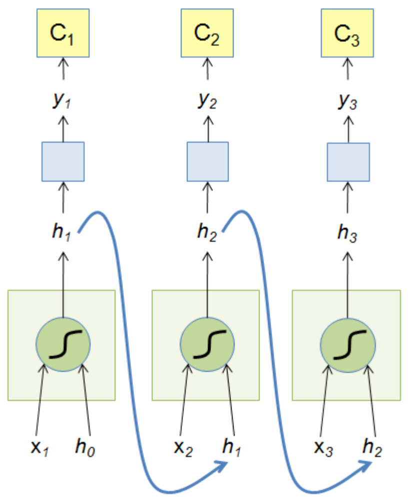


Figure 9.11: A schematic representation of the Vanilla RNN Cell, unrolled in time.

The equations that govern the forward pass through a Vanilla RNN cell are given by:

$$\vec{h}_{t+1} = \sigma(A\vec{h}_t + B\vec{x}_{t+1} + b);$$

$$\vec{y}_t = \sigma(C\vec{h}_t + D\vec{x}_t + \tilde{b}),$$

where A, B, C, D are the associated weights, b and $\tilde{b}$ are neuron biases, and σ indicates the non-linear activation function.

During training, the parameters of the RNN are updated via *Back-Propagation Through Time* (BPTT), typically using Stochastic Gradient Descent [Bot10].

9.7.1 Drawbacks of Vanilla RNNs

Despite the simplicity of Vanilla RNNs, they suffered from several drawbacks. Three of the principal shortcomings were:

- First, multiple chained Vanilla RNNs, led to problem of gradients vanishing out to zero, while training.
- The second drawback was that they essentially tended to retain the cell states from the recent past only.
- Finally, the weighting between the current input and the previous hidden state remained constant over time in a Vanilla RNN cell and is not adaptive.

9.8 Long Short-Term Memory Networks (LSTMs)

Long Short-Term Memory Networks were introduced to address the above mentioned complications, typical of Vanilla RNNs [GSC99, LBE15].

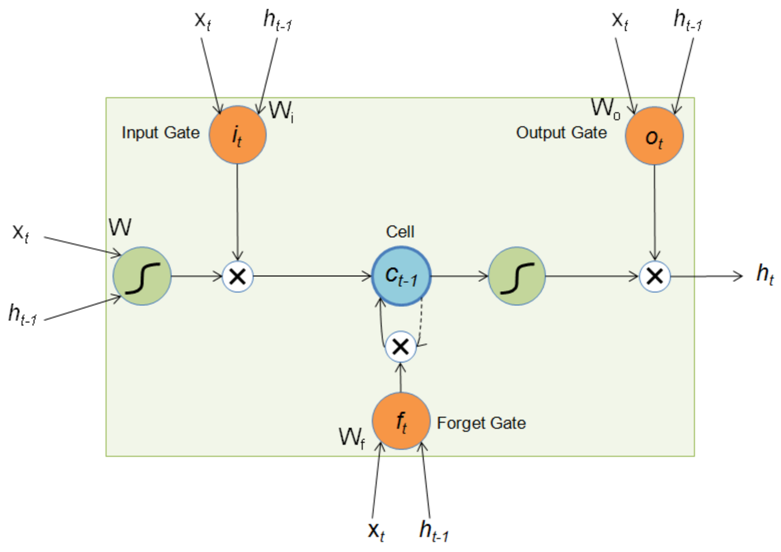


Figure 9.12: A schematic representation of the LSTM Cell.

The key idea in an LSTM cell is to allow a cell state to bypass, the influence of the input at a certain time-step, if necessary. This objective is accomplished by introducing three gates in the cell, viz:

- *Forget Gate*: Controls what fraction of the previous cell state should be retained.
- *Input Gate*: Controls what fraction of the current input should be used in the forward pass.
- *Output Gate*: Controls what fraction of the output should be exposed beyond the cell.

These different gates allow for "skip-connection" like edges in the RNN cell. The following equations, that characterize the behavior of an LSTM cell, bring this out clearly (symbols have their usual meaning).

$$
\begin{aligned}
i^t &= \sigma_i(W_{ix}x^t + W_{ih}h^{t-1} + w_{bi}),\\
f^t &= \sigma_f(W_{fx}x^t + W_{fh}h^{t-1} + w_{bf})\\
o^t &= \sigma_o(W_{ox}x^t + W_{oh}h^{t-1} + w_{bo})\\
\tilde{c}^t &= \sigma_c(W_{cx}x^t + W_{ch}h^{t-1} + w_{bc})\\
c^t &= f^t \circ c^{t-1} + i^t \circ \tilde{c}^t\\
h^t &= o^t \circ \sigma_h(c^t)
\end{aligned}
$$

The first three equations, control the input, forget, and output gates, while the fourth and the fifth ones control the cell state at the next time-step.

9.8.1 Drawbacks of the LSTM

Despite the power of the LSTM cell, they have a large number of parameters and are thus often susceptible to over-fitting. This makes LSTM units harder to train.

9.9 Gated Recurrent Units (GRUs)

In order to fix the challenges arising out of the difficulty of training LSTM units, *Gated Recurrent Units* (GRUs) were introduced [CGCB14].

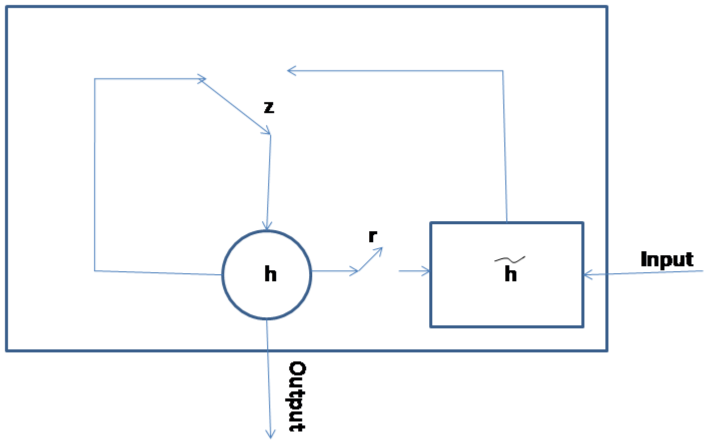


Figure 9.13: A schematic representation of the GRU Cell.

Figure 9.13 shows a representative figure of the GRU cell. These networks essentially operate just like a LSTM cell. However, as is shown in the figure, here we have two gates instead of three and thus we have fewer parameters, viz. the *Input Gate* and the *Forget Gate*.

The following equations characterize the working of the two gates of the GRU cell (where the symbols have their usual meaning).

$$z^t = \sigma(U_Z x^t + W_Z h^{t-1} + b_z),$$

$$r^t = \sigma(U_r x^t + W_r h^{t-1} + b_r)$$

The state updates are governed by the following equations:

$$\tilde{h}^t = \sigma(U_h x^t + W_h(h^{t-1} \odot r_t) + b_h)$$

$$h^t = (h^{t-1} \odot z_t) + (\tilde{h}^t \odot (1 - z_t))$$

Given the fewer parameters of a GRU cell, they are easier to train and thus have been extensively used in recent research, such as sentiment classification [TQL15].

In this class, we thus explored an assortment of neural network architectures for modeling data in a discriminative fashion.

Bibliography

[AJM15] Ejaz Ahmed, Michael Jones, and Tim K Marks. An improved deep learning architecture for person re-identification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3908–3916, 2015.

[ali] Activation Function. https://en.wikipedia.org/wiki/Activation_function. [Online; accessed 2-October-2017].

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[FMI83] Kunihiko Fukushima, Sei Miyake, and Takayuki Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, (5):826–834, 1983.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[GEB16] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2414–2423, 2016.

[GSC99] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[Hop82] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[knn] Neural Networks, Manifolds, and Topology. http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/. [Online; accessed 4-October-2017].

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LBD+90] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.

[LBE15] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[Ros58] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[TQL15] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification. In *EMNLP*, pages 1422–1432, 2015.

[YSXZ15] Wenpeng Yin, Hinrich Schütze, Bing Xiang, and Bowen Zhou. Abcnn: Attention-based convolutional neural network for modeling sentence pairs. *arXiv preprint arXiv:1512.05193*, 2015.

[zoo] The Neural Network Zoo. http://www.asimovinstitute.org/neural-network-zoo/. [Online; accessed 4-October-2017].