AA 274: Principles of Robotic Autonomy — Winter 2018

Lecture 9: Machine Learning and Modern Visual Recognition Techniques

Scribes: Chi Zhang, Honghao Wei, Matthew Tan, Taylor Howell, Brian Jackson, Saifan Rafiq

9.1 Machine Learning Overview

9.1.1 Supervised vs. Unsupervised Learning

Machine learning is a field of computer science that gives computers the ability to learn without being explicitly programmed [1]. Problems in this field can generally be categorized as either supervised or unsupervised learning. The goal of supervised learning is to take a set of labeled training data and learn the function that best predicts the label of a new, previously unseen, input. Classic examples include image classification (i.e. whether or not a picture contains a cat) or predicting the selling prices of houses as a function of various features, such as location, number of bedrooms, square footage, etc. The goal of unsupervised learning, on the other hand, is to find patterns or structure in a set of unlabeled data, such as image compression. These types of learning can be defined more formally:

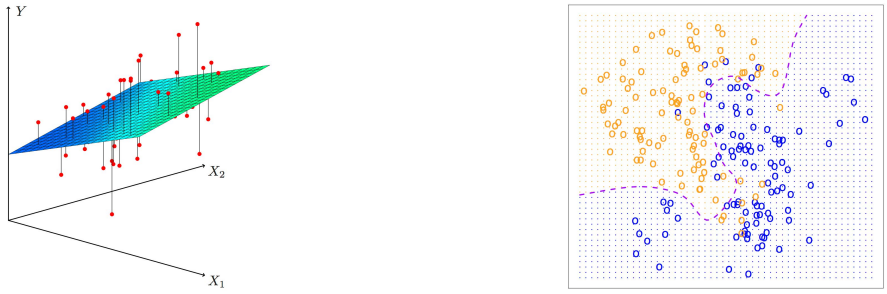
- **Supervised learning (classification, regression)**:
  Given $(x^1, y^1), ..., (x^n, y^n)$ choose a function $f(x) = y$
  x_i = data point
  y_i = class/value/label
- **Unsupervised learning (clustering, dimensionality reduction)**:
  Given $(x^1, x^2, ..., x^n)$ find patterns in the data

9.1.2 Types of Supervised Learning

Here we focus our attention on supervised learning. Supervised learning can generally be split into two categories: classification and regression. In classification problems the goal is to output a discrete value, such as a 1 or 0 (whether or not the picture is a cat). Regression, on the other hand, attempts to learn a function that returns values in the continuous domain (such as house prices).

In either type of problem, the goal is to learn the function $f(x)$ that best approximates the data. In order to evaluate how well a model approximates the data we define a loss (or cost) function that, when minimized, provides the optimal model parameters. There are many types of loss functions, but here we present a few of the most common for both regression and classification:

- **Regression (see Figure 9.1a)**
  ℓ^2 loss: $\sum_i |f(x^i) - y^i|^2$: Assigns a higher penalty to outliers
  ℓ^1 loss: $\sum_i |f(x^i) - y^i|$: Applies equal penalty to outliers
- **Classification (see Figure 9.1b)**
  $0-1$ loss: $\sum_i \mathbf{1}\{f(x^i) \neq y^i\}$: Non-differentiable
  Cross entropy loss: -$\sum_i (y^i)^T \log(f(x^i))$: Differentiable version of the $0-1$ loss

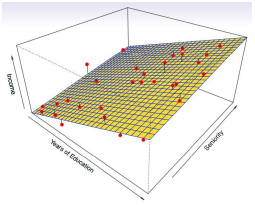


(a) Regression (b) Classification

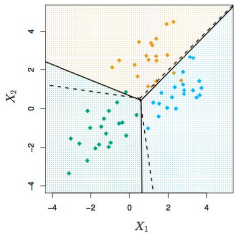
Figure 9.1: Examples of regression and classification for supervised learning

9.1.3 Parametric vs. Non-parametric Models

Within machine learning models can also be defined as being either parametric or non-parametric models. Parametric models can entirely described by a set of learned parameters learned from training data. Once the parameters are learning (using gradient descent, for example), the model no longer needs access to the original training data. The expressivity of the model is therefore a direct result of the feature set and the model parameters. Non-parametric models, however, learn models that are dependent upon the original training data. K-nearest Neighbors, for example (an unsupervised learning algorithm) uses the actual data to categorize any additional data points, and is therefore non-parametric. Examples are shown in Figures 9.2 and 9.3 below.

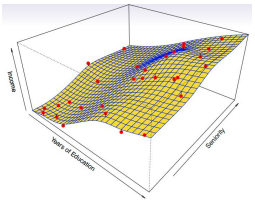


Linear regression

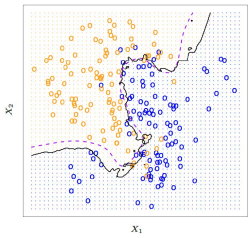


Linear Classifier

Figure 9.2: Examples of parametric models



Spline fitting



k-Nearest Neighbors

Figure 9.3: Examples of non-parametric models

9.1.4 Finding the Optimal Model

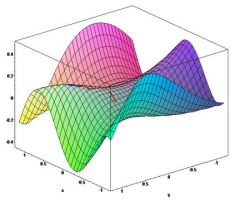
Once we define a model and a loss function, the final step is to find the best model parameters that minimize the loss. Sometimes an analytical solution can be found, such as with linear least squares:

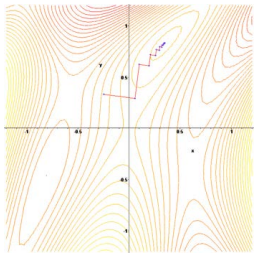
$$\begin{bmatrix} y_1^1 & y_2^1 \\ y_1^2 & y_2^2 \\ \vdots & \\ y_1^n & y_2^n \end{bmatrix} = \begin{bmatrix} x_1^1 & x_2^1 & \dots & x_k^1 \\ x_1^2 & x_2^2 & \dots & x_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_2^n & \dots & x_k^n \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{11} & a_{12} \\ \vdots & \vdots \\ a_{k1} & a_{k2} \end{bmatrix}$$

$$f_A(x) = xA, \ell^2 loss$$

$$\hat{A} = (X^T X)^{-1} X^T Y$$

However, finding an analytical solution is often extremely difficult or computationally inefficient. In practice, machine learning relies on the relatively simple but powerful numerical optimization algorithm of gradient descent. Instead of updating parameters of the model by trial-and-error, the parameters are updated in the direction of greatest change (i.e., along the gradient).





Typically, we then define a cost function J, which a combination of the loss function for each datapoint in the dataset:

$$J = \frac{1}{n} \sum_{i=1}^{n} |f(x^i) - y^i|^2 = \frac{1}{n} \sum_{i=1}^{n} L_i$$

However, computing ∇L could be very computationally intensive as it is not uncommon for training sets to have millions of samples. Alternatively, we can form an approximation using part, or a batch, of the training set. This technique is known as stochastic gradient descent and in practice works quite well.

$$\nabla L \approx \frac{1}{|S|} \sum_{i \in S \subset \{1,\dots,n\}} \nabla L_i$$

Figure 9.5: Examples of stochastic gradient descent. For "batch" gradient descent where the entire dataset is processed at the same time, the cost function J should monotoncially decrease with each step. Stochastic gradient descent, however, will move in random directions but with an overall tendency towards the minimum. Stochastic gradient descent is not guaranteed to converge, but often does in practice. Often the use of "mini-batches" is used to balance the computational benefits of stochastic gradient descent and the stability of "batch" gradient descent, where small "batches" of the training set are processed at the same time (typically 32, 64, or 128). Even better performance can be achieved by using more advanced algorithms such as gradient descent with momentum, RMSprop, or the Adam Optimizer that combines these two.

9.1.5 Regularization

When the algorithm performs well on the training set but poorly in the real world (typically represented by a hold-out cross-validation set of unseen examples), we say the model is "overfitting" the training data. In other words, the model has learned the particular characteristics of the training dataset so well that it fits the noise as well as the general trend. To avoid overfitting the model to the training data (we want our model to generalize well to new samples), we may add additional terms to the loss function to penalize "model complexity". This is known as regularization.

- ℓ^2 regularization: $||A||_2$ often corresponds to a Gaussian prior on parameters A.
- ℓ^1 regularization: $||A||_1$ often encourages sparsity in A (easier to interpret/explain).

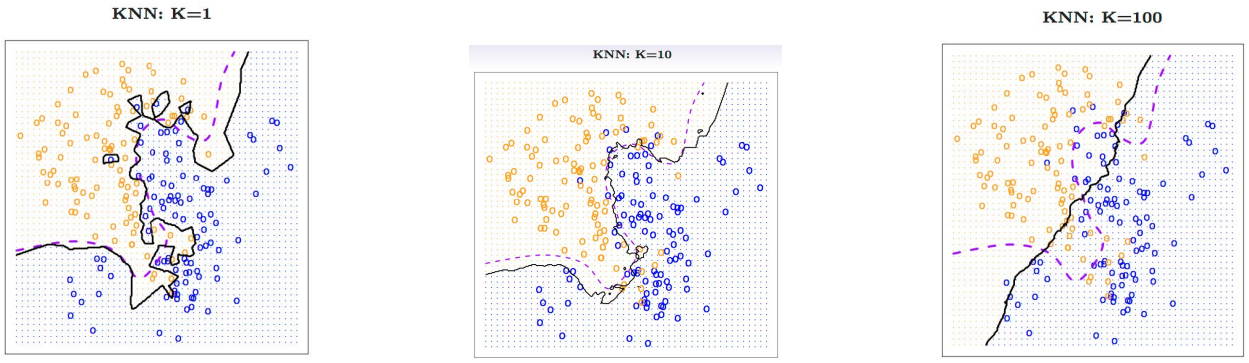


Figure 9.6: Examples of regularization. When the regularization is too low or non-existent we see behavior shown in the figure on the left, where the model is fitting the "noise" of the data, and is clearly not generalizable (we call this high variance). When regularization is too high, as shown in the figure on the right, the model effectively reduces in complexity and approaches a simple linear classifier, and performs poorly on both the training and test datasets (we call this high bias). The figure in the middle shows a good balance between these extremes.

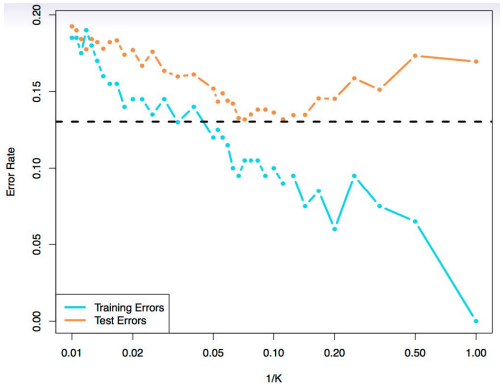


Figure 9.7: Training and test set error. A typical plot for a model that over fits to the training set at the expense of generalization. As the model's performance on the training set increases, the performs on a new set of examples gets worse. Regularization helps overcome this behavior.

9.1.6 Interpretation of Linear classifiers

Linear models of the form $f(x_i, W, B) = Wx + b$ are commonly used in machine learning. Intuitively, we can think of this as assigning a weight to each feature (plus a bias term). For example, in predicting house prices we may believe that the number of rooms in a house is more important to the final selling price than the number of light switches, so we would assign more weight to the former feature. This same idea can be applied to each pixel value in an image to determine whether it is a cat.

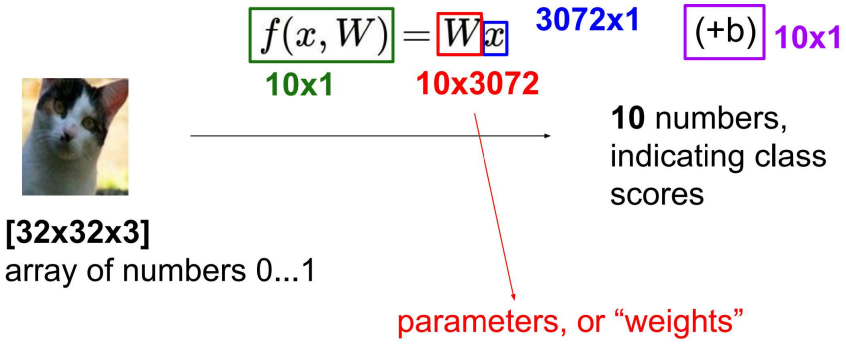


Figure 9.8: Using a linear classifier to classify an image. An image is typically a 3D matrix with height, width, and depth (having 3 colors–RGB). We can reshape a $hxwxd$ matrix to a vector of length $h * w * d$ and give the image class scores using a weighting matrix W an bias vector b. The number of rows in W and b corresponds to the number of classes we are evaluating.

To learn the parameters of a (linear) model, large datasets are used. One example is the CIFAR-10 dataset which contains ten classes of images, each with 6000 examples. After a classifier is trained on such a dataset, each row of W can be thought of as a "template" for nearest neighbor classification.

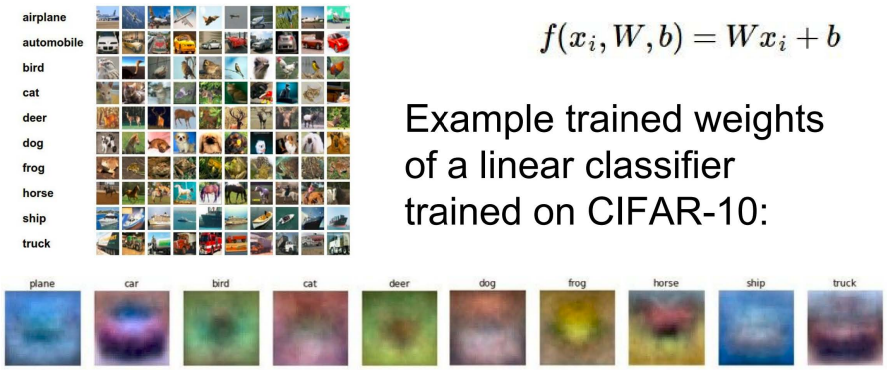
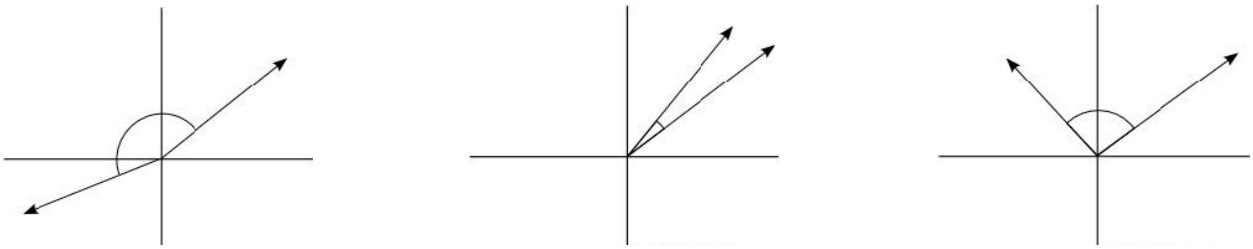


Figure 9.9: CIFAR-10 dataset example

Continuing with the "template" idea, another way to think about linear classifiers is as dot products, which can be thought of as a measure of similarity. By padding the feature vector x with a 1, we can include the bias term in our weights such that $f(x_i, W) = Wx$. The dot product of the class template with our image that produces the largest value should be the class of our image.

$$\omega^T x = ||\omega|| ||x|| cos(\theta) \rightarrow cos(\theta) = \frac{\omega^T x}{||\omega|| ||x||}$$

We can also interpret the dot product as a projection: "how much" ω is in x according to the component x along ω.



$cos(\theta) \approx -1$ $cos(\theta) \approx 1$ $cos(\theta) \approx 0$

Figure 9.10: Cosine Similarity. The extrema are when two vectors are nearly opposite, very similar, or have limited project.

9.1.7 Softmax Regression

Once we have found class scores it's often useful to normalize them so classes can be more easily compared. Outputs from the model can be turned into a probability vector over classes using the softmax function. The sum of all class scores will sum to 100%.

$$p(y^i = j|x^i) = \frac{e^{x^i W_j + b_j}}{\sum_k e^{x^i W_k + b_k}}, \sigma(z) = \begin{bmatrix} \frac{e^{z_1}}{\sum_k e^{z_k}} \\ \vdots \\ \frac{e^{z_m}}{\sum_k e^{z_k}} \end{bmatrix} \tag{9.1}$$

Another common form for class scores comes from Cross-entropy loss computed against "one-hot" class vector $[0, ..., 1, ..., 0]$ where all but the correct class are assigned 0 and the correct class is assigned 1.

9.1.8 Generalizing linear models - Features

Linear regression/classification can be very powerful when empowered by the right features. While the mechanics of the classifier remain linear, the features can be nonlinear via basis functions. For example, a nonlinear feature may be a modified original feature: x^n, $x_i x_j$, etc. For example, our original features may be voltage and current from a circuit, but the product of these is a new feature: power, that may be more informative to our model.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ 1 & x_3 & x_3^2 & \dots & x_3^m \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}$$

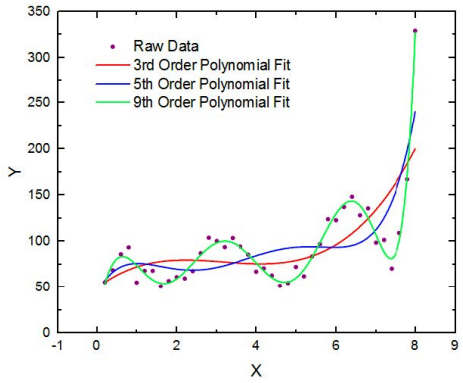


Figure 9.11: Nonlinearity via basis functions. Higher order models can be fit using a linear model when nonlinear features are used.

One example of using the "right features" is demonstrated by eigenfaces, a means for facial recognition. Instead of classifying unprocessed images of faces, eigenvectors from a set of images can be used to reduce the dimensionality of the feature space of an image for classification.

Figure 9.12: Eigenfaces

Another very successful feature set for shape classification/regression is Histogram of Oriented Gradients (HOG). These human-designed features are quite successful at finding the outline of shapes by calculating the gradients of pixels.

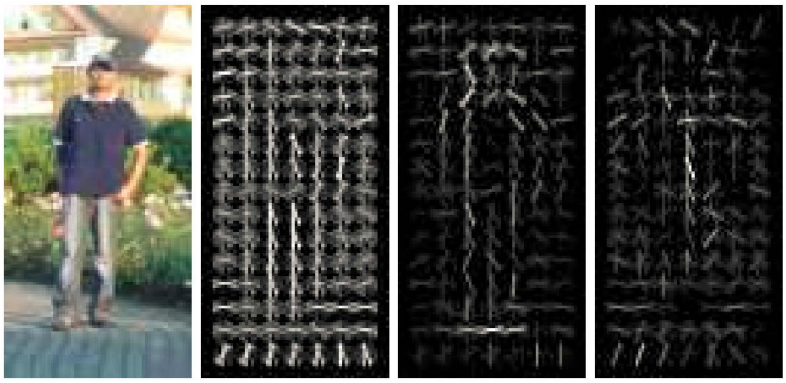


Figure 9.13: (Dalal and Triggs 2005). Left to right: test image, visualization of HOG descriptor, HOG descriptor scaled by positive weights, HOG descriptor scaled by negative weights.

While humans have been successful at feature extraction across many applications, it is possible to automatically extract features using neural networks that rely on gradient descent.

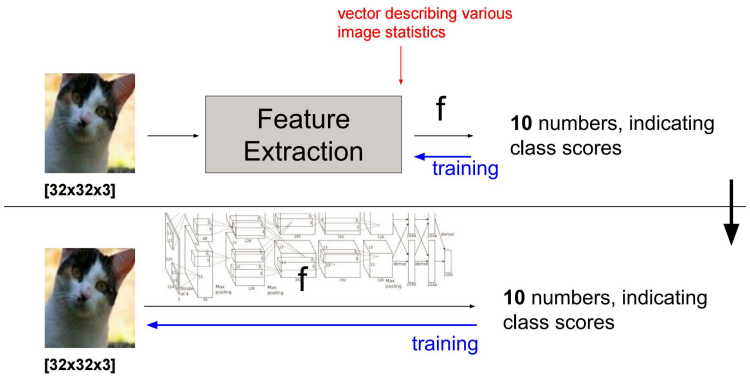


Figure 9.14: Comparison of classification using bespoke feature extraction vs end-end learning

9.2 Neural Network Basics

Work on Artificial Neural Networks, generally known as just 'Neural Networks', has been inspired by the fact that the human brain processes information in a completely different way than computers. Due to this, human brains are much more efficient at certain tasks, such as vision, language processing, etc. than computers. Artificial neural networks aim to mimic this biological model by employing a large number of simple interconnected processing units or 'neurons'. We may thus offer the following definition of a neural network viewed as an adaptive machine[2]:

A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:
1. Knowledge is acquired by the network from its environment through a learning process.
2. Inter-neuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

9.2.1 Perceptron - Analogy to a Biological Neuron

The figure below shows a simplified diagram of a biological neuron:

There are approximately 86 billion neurons in the human nervous system and they are connected with approximately 10^14 - 10^15 synapses[3].

The figure below shows a commonly used mathematical model of a neuron:

Signals travel along the axons (e.g. x0) and interact multiplicatively (e.g. w0x0) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitatory (positive weight) or inhibitory (negative weight)) of one neuron on another. We can model the firing rate of the neuron with an activation function f, which represents the frequency of the spikes along the axon. Historically, a common choice of activation function was the sigmoid function, but the ReLU function is more commonly used nowadays. This is because the gradient of the sigmoid function becomes small as its value increases which reduces the training speed of the model.

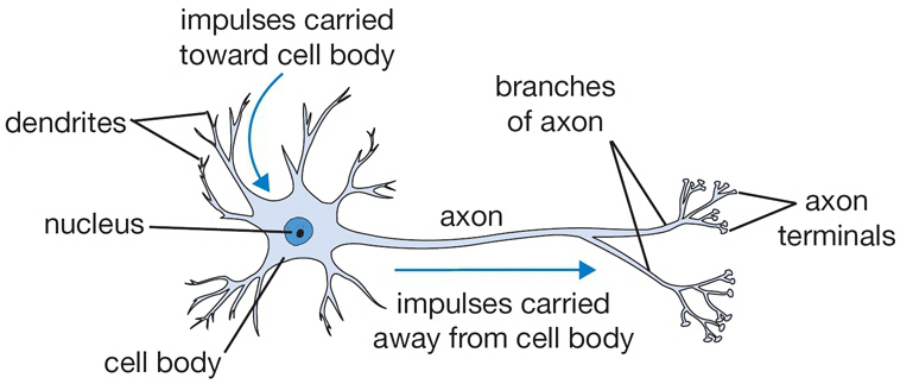


Figure 9.15: Simplified model of a biological neuron[3]

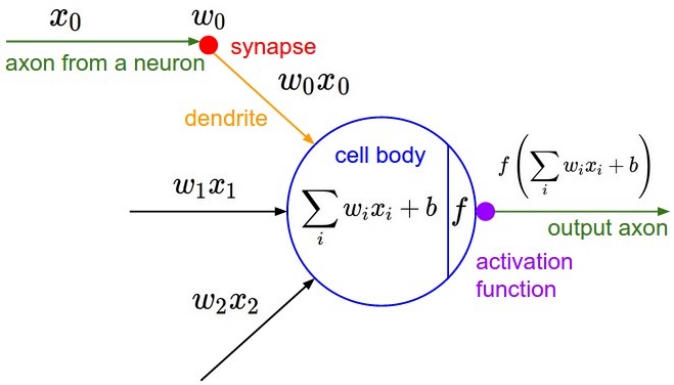


Figure 9.16: Mathematical model of a neuron[3]

9.2.2 Single Layer Network / Logistic Regression

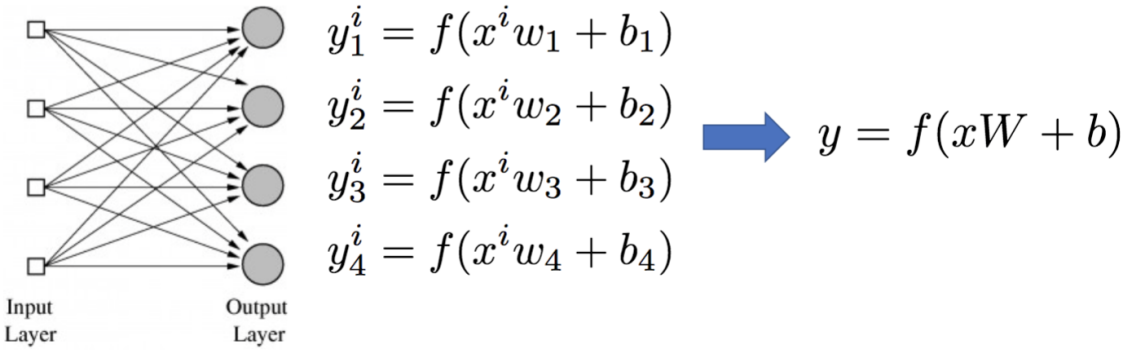
Inspired by this model of the neuron, a 1 layer neural net can be built which takes in binary inputs and provides binary output by taking a weighted sum and passing it through an activation function (as seen below). The theory is that if these weights are tuned perfectly, then it should be able to classify the inputs correctly.

9.2.3 Multi-layer Neural Network / Deep Learning

We can get more resolution on our classifications by putting multiple of these layers together. In such regard, each layer can "learn" to classify a different part of the input. Take for an example the problem of classifying handwritten digits, in particular classifying the number 3. The first layer can be responsible for classifying a curved top, the second a curved bottom, etc.

A sample is shown below (from : http://www.cs.cmu.edu/ aharley/vis/conv/flat.html)

Representing this is simply a bunch of single layer neural networks linked together where the output of 1 layer is the input of the second layer.



$$y_1^i = f(x^i w_1 + b_1)$$
$$y_2^i = f(x^i w_2 + b_2)$$
$$y_3^i = f(x^i w_3 + b_3)$$
$$y_4^i = f(x^i w_4 + b_4)$$
$$\Rightarrow \quad y = f(xW + b)$$

Figure 9.17: Single layer neural net[**?**]

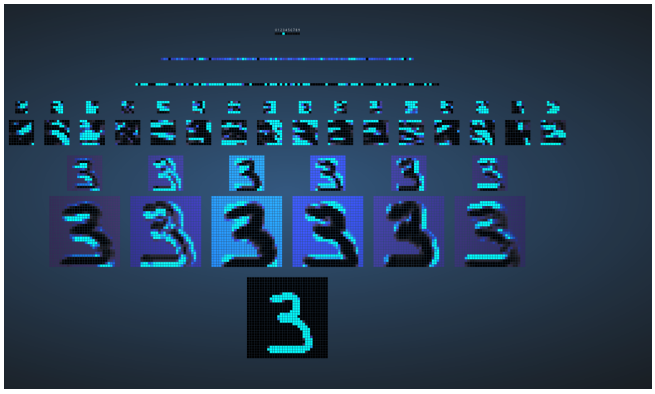
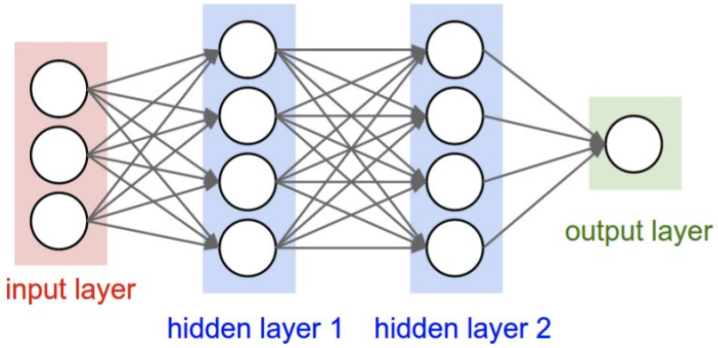


Figure 9.18: Deep Neural Net for Classifying Handwritten Numbers[]



$$h_1 = f_1(xW_1 + b_1)$$
$$h_2 = f_2(h_1 W_2 + b_2)$$
$$y = f_3(h_2 W_3 + b_3)$$

Figure 9.19: Deep Learning[]

9.2.4 Activation Functions

If we didn't use non-linear activation functions, the output of the neural network would just be a linear function of the input. Such a network is basically just a Linear regression Model and and no matter how many layer and nodes we add to such a network, it is equivalent to having a network with a single node.

Hence, we use Activation functions to make the network more powerful give it the ability to represent non-linear complex functional mappings between inputs and outputs.

Another important feature of an Activation function is that it should be differentiable. This is because we perform back-propagation to calculate the gradients of the Loss function with respect to the weights and then accordingly optimize weights using gradient descend or any other Optimization technique to decrease the Loss Function.

The following are the most commonly used activation functions:

1. **Sigmoid Function:** The Sigmoid Function takes a real-valued number and maps it to a range between 0 and 1.

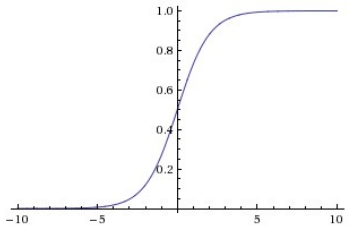


Figure 9.20: The sigmoid function[3]

$$f(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid Activation functions have historically been used extensively but have now fallen out of favor due to the following drawbacks:

   (a) They are prone to saturate as the gradients are very small away from the center. When this occurs, almost no signal will flow through the neuron to its weights and recursively to its data.

   (b) They are not zero-centered.

2. **Tanh function:** The Tanh function looks quite similar to a sigmoid function. It maps a real-valued number to a range between -1 and 1.

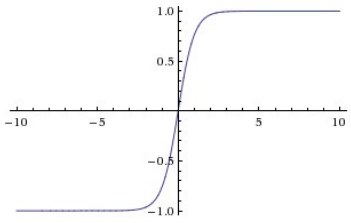


Figure 9.21: The tanh function[3]

$$f(x) = \tanh(x)$$

It saturates in the same way as a sigmoid does but it has the advantage of being zero-centered. Hence, in practice, it is always preferred to a sigmoid function.

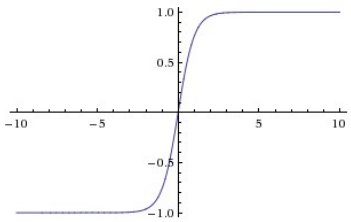


Figure 9.22: The ReLU function[3]

3. **ReLU Function:** The Rectified Linear Unit is simply a threshold at zero and has become very popular in the last few years.
$$f(x) = max(0, x)$$
It has the following advantages:
   (a) It greatly accelerates the convergence of stochastic gradient descent compared to tanh/sigmoid.
   (b) It is computationally cheaper to implement than tanh/sigmoid as it can implemented by simply thresholding a matrix of activations at zero.

   The main disadvantage of the ReLU is that ReLU units can irreversibly die during training since they can get knocked off the data manifold.

4. **Leaky ReLU Function:** The Leaky ReLU attempts to fix the problem of dying nuerons by providing a small negative slope in the $x < 0$ region.

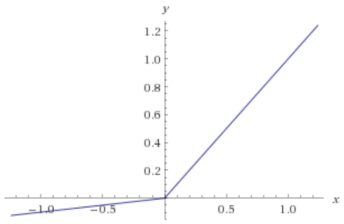


Figure 9.23: The Leaky ReLU function[3]

$$f(x) = max(0.1x, x)$$

9.2.5 Training Neural Networks

We now need a way to teach these neural networks weights to make correct predictions. This is done by running the network on known data points and updating the weights of the neurons based on the correctness of the model in classifying the input.

Running this however on each input one at a time can take a long period of time for training. Instead these models are run on a sample batch of data. In particular the steps are the following.

1. Sample a batch of data. We can use a smaller batch of data at each iteration to train the model to decrease training time.

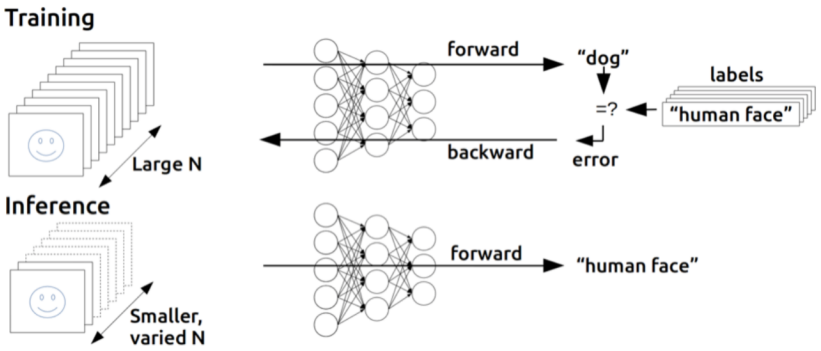


Figure 9.24: Training a neural net[]

2. Run the input it through the graph and compute the loss. The loss function provides serves as a quality metric and provides us an idea on the error of our model. That is, how correct it is.

3. Backpropagate Activation functions are chosen to be differentiable to make this step easier. Backpropagation is done by taking the gradient with respect to the weights of each neuron in each layer.

4. Update these parameters using SGD (Stochastic Gradient Descent) Update each parameter using the gradient calculated in the previous step

5. Repeat many times

9.2.6 Overfitting

Creating complex models sometimes have the disadvantage of being extremely accurate only on the test data. That is, the model is able to classify its entire training data with high accuracy however is unable to generalize and fails with test / real data.

On the other hand too simple models are unable to learn the more subtle trends in the data.

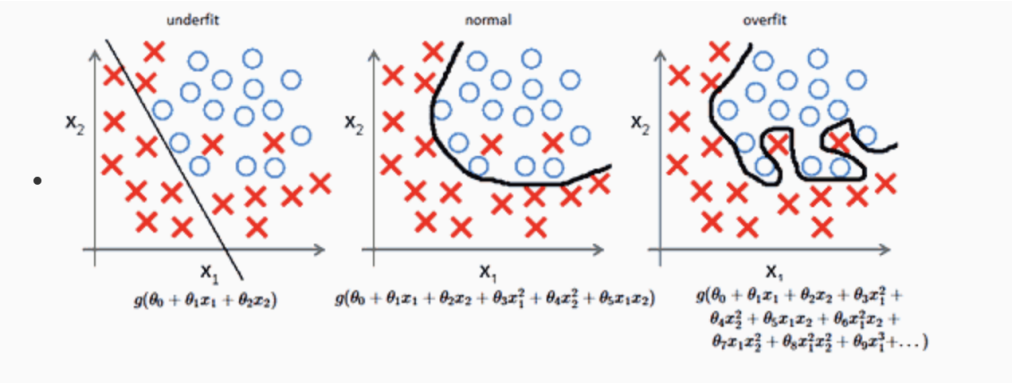


Figure 9.25: Overfitting[]

http://mlwiki.org/index.php/Overfitting

In the example above the left model was too simple and was unable to classify some of the values correctly.

On the other the right model despite perfectly classifying on this training dataset is unlikely to perform as well in real-world situations. The middle offers a good classification.

To combat overfitting, regularizaiton is usually introduced in models. Regularization is a technique that penalizes complex models or prevents them from being made. Some of these techniques include

Dropout Randomly select neurons that will not be activated at each pass during training.

L2 Regularization Penalize the square magnitude of all parameters which forces neuron size to be smaller.

Max Norm Constraints Limit the maximum weight of a neuron.

9.3 Convolutional Neural Networks

A simple Convolutional Neural Networks is a sequence of layers, and every layer of a Convolutional Neural Networks transforms one volume of activations to another through a differentiable function. There are four main types of layers to build ConvNet architectures: **Convolutional Layer**, **Nonlinearity Layer**, **Pooling Layer**, and **Fully-Connected Layer**. These layers are stacked to form a full ConvNet architecture. In this way, Convolutional Neural Networks transform the original image layer by layer from the original pixel values to the final class scores.

In this section, we mainly discuss the three types of layer in Convolutional Neural Networks, including Convolutional layer, Pooling layer and Fully-connected layer.

9.3.1 Convolutional Layer

Parameter Sharing Parameter sharing scheme is used in Convolutional Layers to control the number of parameters. It is hard to make perceptrons scalable when the input image size is large given the fact the number of parameters grow quickly with the input size. However, we can dramatically reduce the number of parameters by making one reasonable assumption. If we know the input is image data, we can assume some spatial symmetries. In other word, if one feature is useful to compute at some spatial position (x, y), then it should also be useful to compute at a different position (x_2, y_2), so same parameters could be used.

Convolution Details The convolution operation essentially performs dot products between the filters and local regions of the input. In other words, each element is computed by element-wise multiplying the local regions of the input (blue part in Figure 9.26) with the filter (red part in Figure 9.26), summing it up, and then offsetting the result by the bias. The output is the green part in Figure 9.26.

Noticing we give the demo in 2D format. In real cases, the input would be a 3D tensor while he depth of the output volume is a hyperparameter. It corresponds to the number of filters we would like to use, each learning to look for something different in the input. Take Figure 9.27 as an example, if we have 6 5×5 filters, we will get 6 separate activation maps as output.

Spatial arrangement Besides depth, there are two hyperparameters to control the size of the output volume: the stride and zero-padding

- **stride** Stride specifies how we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around.
- **zero-padding** In some cases, the size of filters and stride dont fit neatly and symmetrically across the input. For example, if the input size $H = W = 10$ while the filter size is $F = 3$ and it takes stride

$S = 3$, it would be impossible to use stride $S = 2$, since

$$\begin{aligned}(H - F)/S + 1 &= (10 - 3)/3 + 1 = 3.33 \text{ (Not an integer)} \\ (W - F)/S + 1 &= (10 - 3)/3 + 1 = 3.33 \text{ (Not an integer)}\end{aligned} \tag{9.2}$$

Therefore, we need to use zero-padding. We add additional zero-paddings in the contour of the image. If we take zero-padding $P = 1$, we would have

$$\begin{aligned}(H - F + 2 * P)/S + 1 &= (10 - 3 + 2 * 1)/3 + 1 = 4 \text{ (An integer)} \\ (W - F + 2 * P)/S + 1 &= (10 - 3 + 2 * 1)/3 + 1 = 4 \text{ (An integer)}\end{aligned} \tag{9.3}$$

In general, setting zero padding to be $P = (F1)/2$ when the stride is $S = 1$ ensures that the input volume and output volume will have the same size spatially. It is very common to use zero-padding in this way.

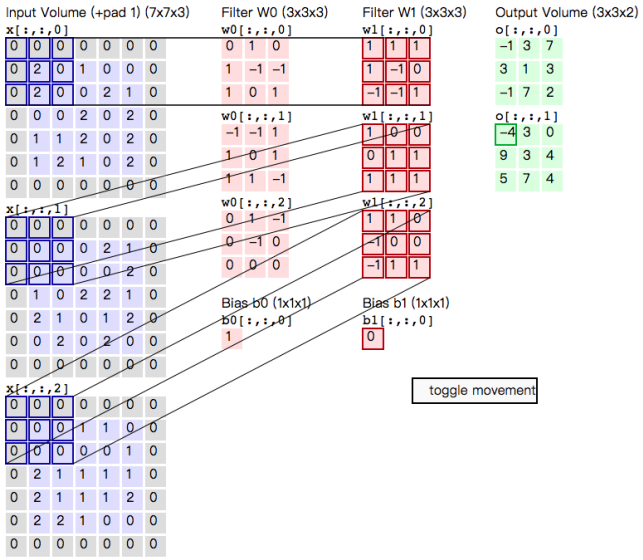


Figure 9.26: Convolution Demo[3]

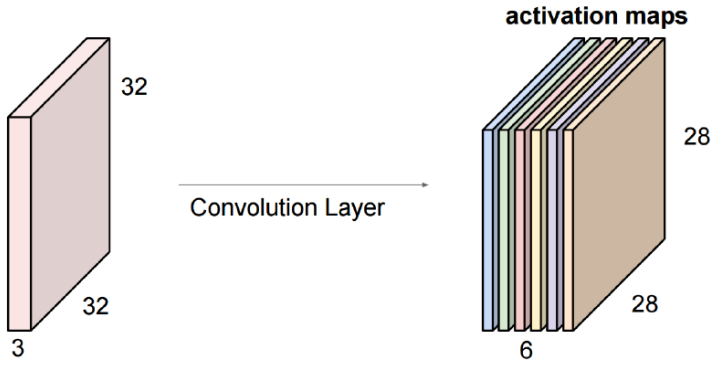
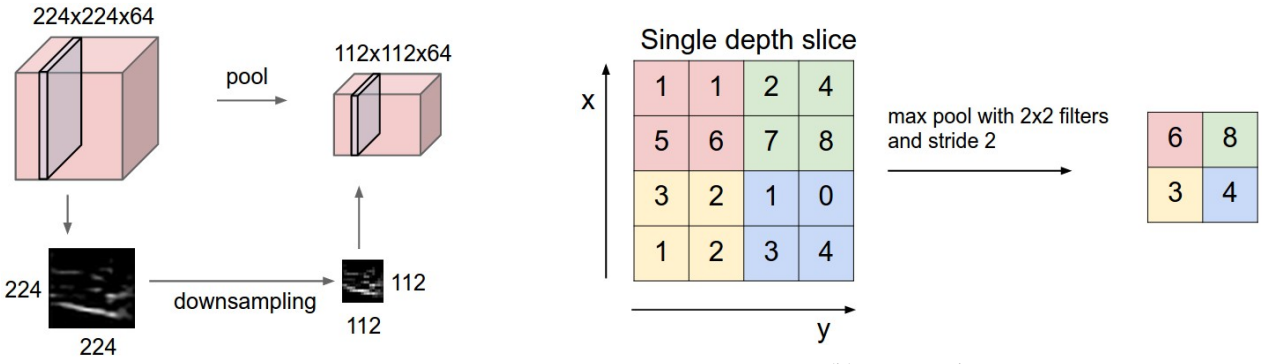


Figure 9.27: Number of filter and depth of output[3]

9.3.2 Pooling Layer

In practice, it is common to periodically insert a Pooling Layer in-between successive Conv layers in a ConvNet architecture. It is mainly used to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation load, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation if max pooling, AVG if average pooling, etc. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. In this case, every pooling operation would be taking a max or average over 4 numbers (little 2×2 region in some depth slice). The depth dimension remains unchanged. In a nutshell, *pooling layer* ***downsamples*** *the volume spatially, independently in* ***each*** *depth slice of the input volume.*



(a) general pooling

(b) max pooling

Figure 9.28: Pooling Layer[3]

The Fig.9.28a shows how pooling works generally. A 2×2 pooling filter with stride 2 is applied on the $224 \times 224 \times 64$ input volume, yielding the $112 \times 112 \times 64$ output volume. Note that the width and height of the input volume shrinks to half while the depth is preserved. The most common downsampling operation is MAX. The Fig.9.28b here shows max pooling with a stride of 2. That is, each max is taken over 4 numbers in a 2×2 square. For instance, the top-left red square squashes to a single value, `6`, which is $\texttt{max}\{\texttt{1, 1, 5, 6}\}$.

9.3.3 Fully-Connected Layer

The last few layers of a ConvNet architecture are typically Fully-Connected (FC) Layers. As seen in regular neural networks, FC Layers serve as a linear classifier for classification or regression. Their activations can be computed with a matrix multiplication followed by a bias offset, *i.e.* $f = Wx + b$. See the Neural Network Basics section of the note for more information.

9.3.4 Mainstream Model Architectures

Empowered with impressive ability of feature extraction and optimization, deep convolutional neural networks has become the cornerstone of data-driven visual recognition techniques nowadays. Since AlexNet's overwhelming success in ILSVRC 2012, a number of milestone model architectures have been proposed. The most common are:

- **AlexNet**[4]: Developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The AlexNet significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error) ILSVRC challenge in 2012. It is the first work that popularized Convolutional Neural Networks in Computer Vision.
- **VGGNet**[5]: Proposed by Karen Simonyan and Andrew Zisserman. The VGGNet was the runner-up in ILSVRC 2014. Its main contribution was in showing that the depth of the network is a critical component for good performance. Researchers and engineers tend to design deeper rather than shallower models ever since.
- **GoogLeNet**[6]: As its name shows, it is authored by Szegedy *et al.* from Google. GoogLeNet was the ILSVRC 2014 winner with main contribution in developing Inception Module that dramatically reduced the number of parameters in the network.
- **ResNet**[7]: Developed by Kaiming He *et al.* ResNet was the winner of ILSVRC 2015. It features special skip connections and a heavy use of batch normalization, which is now known as Residual Module. ResNets are currently by far state of the art Convolutional Neural Network models and are the default choice for using ConvNets in practice (as of January 2018).

There also exist more model architectures proposed for specific tasks, such as YOLO[8] and Faster R-CNN[9] in the field of object detection and localization.

References

[1] Machine learning — Wikipedia, the free encyclopedia, 2018. [Online; accessed 12-February-2018].

[2] Simon S. Haykin. *Neural Networks and Learning Machines*. 2009.

[3] Andrej Karpathy. Stanford university cs231n course notes, May 2016. Accessed: 2018-02-07.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[8] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[9] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.