Machine Learning Models and Algorithms Summary

Kent Gauen and Dr. Xiao Wang

Contents

1 Introduction

Machine learning is a powerful tool used in a wide breadth of applications including quality control, image classification, and self-driving cars. We give a broad summary of many fundamental machine learning models, so that the reader may gain an intuition behind their formulation. In this paper, we use the MNIST dataset to demonstrate the mechanisms of machine learning models. There are three major teaching paradigms – supervised, unsupervised, and semi-supervised learning. We will discuss supervied and unsupervised learning. For supervised learning, we discuss logistic regression, support vector machines, and convolutional neural networks. For unsupervised learning, we discuss the k-means cluster method, principal components analysis, and auto-encoders. Each of the methods are separated into the problem formulation, a brief overview of the algorithm. After

each of the models are presented, a summary of the results is provided to compare the performance of the models.

Note that the purpose of this paper is not to provide a comprehensive review of old methodology, introduce new methodology, or present new results. This paper's goal is to present the basics of fundamental machine learning models to a target audience of undergraduate students with a minimal statistics and mathematics background. However, we do expect the reader to be familiar with linear regression.

2 Dataset: MNIST

The MNIST dataset was originally black and white images from the NIST, National Institute of Standards and Technology. They are a mixture of Special Database 1 (high-school students) and Special Database 3 (Census Bureau), which are binary images of handwritten arabic integers from 0 through 9. The MNIST, or Mixed NIST, dataset are normalized to a 20×20 pixel box while preserving their aspect ration. The dataset contains $60,000$ images of digits for training, and $10,000$ images of digits for testing [1].

Figure 1. The MNIST dataset is gray-scale images of handwritten arabic integers, $0-9$. This provides machine learning researchers with an easily understood dataset, demands little pre-processing, and requires a non-trivial solution.

The MNIST dataset is widely popular for benchmarking image classification models. There is little to no pre-processing required, and the task is straight-forward: what is the digit in the image? The task is also non-trivial. For example, how do we construct a model that can recognize a 2 shifted in an image? Linear classifiers struggle classifying shifted images. The MNIST dataset is good choice to use for this paper, given it's educational purpose.

3 Supervised Learning

Classification is the task of determining which category, or class, an input belongs to. Classification tasks inlude any task that categorizes input features into a finite number of classes. A common classification task is image recognition; e.g. a picture's pixels are used as the input, and a model can determine if the image contains a dog, building, car, a combination of two classes, all three classes, or none of them.

In supervised classification, the data set is comprised of N samples of $\mathbf{x}$ written together as $\mathbf{X} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ with target values $\mathbf{t}$, written together as $\mathbf{T} \equiv (\mathbf{t}_1, \ldots, \mathbf{t}_N)$. The input data $\mathbf{x}$ is a column vector containing real valued features with dimensionality $\mathbf{D}$, $\mathbf{x} = (x_1, \ldots, x_D)^T$. The data gives information to predict the target $\mathbf{t}$. While the target value can be written many ways, we will assume the target varaible will be binary column vector of $\mathbf{K}$ dimensions for a k class classification problem, $\mathbf{t} = (t_1, \ldots, t_k)^T$. With the exception of support vector machines, we will pose our classification tasks such that $\sum_{i=1}^{k} t_i = 1$, or in words, each input only has one correct output. Relating to the image recognition task above, we will only consider cases in which an input image contains only a dog, building, car, or none of them.

We use a weight vector $\mathbf{w}$ with dimensions $\mathbf{D}+1$, given our input has dimensions $\mathbf{D}$. The extra dimension for $\mathbf{w}$ is required for a bias term w_0 (not to be confused with *bias* the statistical term). Therefore it is assumed that our weight vectors are defined as $\mathbf{w} = (w_0, w_1, \ldots, w_D)^T$. Our input

features $\mathbf{x}$ are then re-defined as $\mathbf{x} = (1, x_1, \ldots, x_D)^T$. This format allows us to use vectorized equations instead of summations, which will clear-up our equations.

Often we will also use $\phi(\mathbf{x})$ as a basis function for $\mathbf{x}$. Our basis functions can be written as $\phi(\mathbf{x})$ or ϕ, but it is still a function of $\mathbf{x}$ even if the explicit dependence is omitted. It often will be omitted for less cluttered notation.

In the following section we will use the neural network framework to explain (a) multilayer perceptrons, (b) logistic regression, (c) convolutional neural networks, and (d) support vector machines.

3.1 Multilayer Perceptrons Methodoloy & Analysis

Before we discuss neural networks, it is important to motivate why neural networks, and deep learning, are currently a popular topic. In image recognition, many people have studied hand-engineered features to decompose images into fundamental building blocks. These processes include SIFT, GLOH, Spin Image, HoG, Textons, and RIFT. While the results are moderately effective, the process of constructing hand-engineered features is slow. Deep learning is attractive because it allows us to construct better features *automatically*. The most recent milestone is Alex Krizhevsky's 2012 Imagenet competiton victory, which was won by a large margin using deep learning techniques **??**. While we discuss that in more detail later, that paper is an important part of what separated deep learning from machine learning, and what has brought deep learning into vogue.

Neural networks (also called *multi-layer perceptrons*) are the fundamental building block of many modern machine learning models. Let's start with a simple model. If one understands how to analyze a three (or more) layer neural network, the analysis can be applied to arbitrarily sized models. We will look at the four layer network in figure 3.

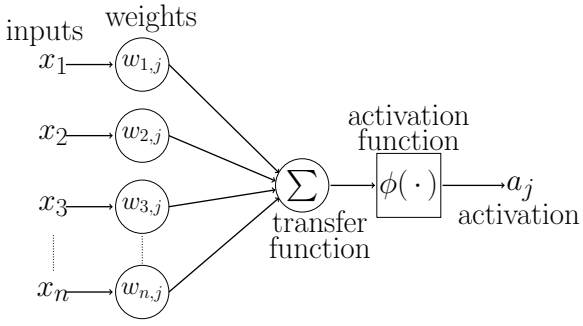

Figure 2. A single node from a neural network. $\sum = w_{0j} + w_{1j}x_1 + w_{2j}x_2 + \ldots + w_{nj}x_n$ and $a_j = \phi(\sum)$, where a_j is the output from the j^{th} node in a layer. The layer is not indicated in this example with indicies.

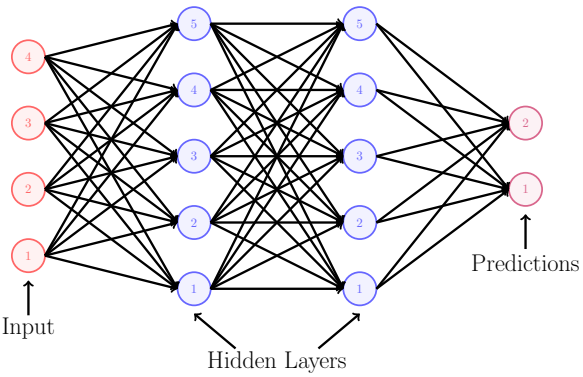

Figure 3. An example neural network with 4 input features, 2 hidden layers with 5 nodes each, and 2 output nodes.

A nerual network is a directed graphical model composed of nodes connecting the output of each node in one layer to each input of the next layer using real valued weights. Each node can be visialized as in figure 2, where $\phi(\cdot)$ denotes an *activation function* and the $\sum$ denotes a summation of the products formed by the dot product, $\mathbf{x}^T\mathbf{w}$. The *activation* of a node, a_j, is given by using the sum of products as the argument for the activation function. Common activiation functions include the $simoid(z)$, $tanh(z)$ and $\text{ReLu}(z)$, and are picked depending on the context. Each node in each layer l has a unique, equal-sized weight vector $\mathbf{w}^l$ where the super-script l denotes that the weights belong to layer $l = \{1, 2, 3\}$. The output from each node can be represented below, where

a_j represents the output from the j^{th} node in a layer.

$$a_j = \phi\left(w_{0j} + w_{1j}x_1 + w_{2j}x_2 + \ldots + w_{nj}x_n\right) \tag{1}$$

The blue nodes in figure 3 are the "hidden nodes", and each layer can be thought of as the output from logistic regression (discussed in next section). The final red outputs can be thought of as the output from multinomial regression (explained in the second section) using the second hidden layer activations as the model inputs. Finally, the red nodes are fed into a cost function (not pictured), that will compute the error between the approximate value y and the target value $\mathbf{t}$. The error, or cost, is a quantitative measure of how well our model performed. The computation of the node activations and the error between the predicted value and our target value is called a *forward pass*.

Now that we can compute a forward pass, let's discuss the *backward pass* of the neural network. The goal of training is to find a set of weights that minimize the cost function. A popular method of finding the optimal set of weights is using back-propogation to iteratively update the weights with *stochastic gradient descent*. This is called training the model.

First consider a linear model. Consider the root-means squared (rmse) cost function we wish to minimize over $\mathbf{w}$. The rmse function is given by:

$$\arg\min_{w} \ \text{RMSE}(\mathbf{x}) = \frac{1}{2}\sum_{\mathbf{x}_i}\left(\mathbf{t}_i - y_i\right)^2 = \frac{1}{2}\sum_{\mathbf{x}_i}\left(\mathbf{t}_i - \mathbf{x}_i^T\mathbf{w}^1\right)^2$$

The gradient of the cost function is the direction of steepest ascent. Since our goal is to minimize the cost function, or equivalently descend along the error surface, we take the negative of the gradient. The negative gradient of the rmse function with respect to the $\mathbf{w}$ is given by:

$$-\nabla\text{RMSE}(\mathbf{x}) = -\frac{\partial}{\partial w}\text{RMSE} = -\sum_{\mathbf{x}_i}\left(\mathbf{t}_i - \mathbf{x}_i^T\mathbf{w}^1\right)\left(\mathbf{x}_i^T\right) \tag{2}$$

This supplies one step of our iterative update. The negative gradient is only locally the steepest descent, so taking small steps ensures that we following snugly along the peaks and valleys of the error surface. Our new weights at time $t+1$ from our old weights at time t, where $\mathbf{w}^1$ is the whole weight vector for layer one, are given by:

$$\mathbf{w}^1_{(t+1)} = \mathbf{w}^1_{(t)} - \eta \ \times \ \nabla\text{RMSE}(\mathbf{x}) \tag{3}$$

The *learning rate* is the "η" variable, and is a hyperparameter set by the user. The reason the weights are updated using a learning rate is to assure that weights are being minimized along the error surface. If η is too large the weights will jump across the error surface, and if η is too small the error will not minimize in finite time. Initialization of the model is another important factor in neural network performance. Figure 4 is an example of what an error surface for a simple neural network might look like, and illustrates the importance of initialization.

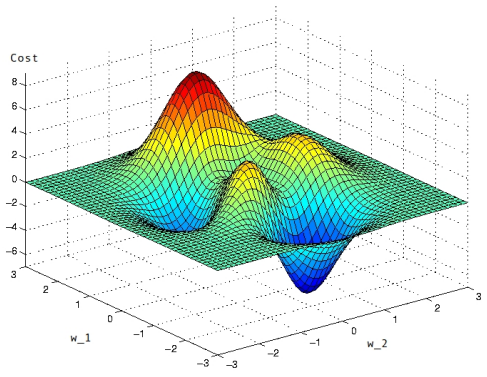

Figure 4. This is an example error surface of which we wish to find the minimum. $C(w)$ is general loss function with model parameters w_1 and w_2. By starting on the deep red peak, we can see that one minima is the light blue local minima directly below the red peak. However, a better solution would be the global minimum in depper blue on the right. This situation illustrates how initialization of model parameters can influence the overall model performance. If we start on the right side of either yellow peak, our minima will be global while the red peak only will reach the local minima. Neural networks often get stuck in local minima, but it is experimentally observed that they are still able to perform very accurately.

Since the learning rate is such an important parameter, stochastic gradient descent is often enhanced with techniques like momentum, weight decay, and learning rate decay. There are many also variations to stocahstic gradient descent that use adaptive learning rates to decrease training time such as adagrad, lbfgs, adamax, rms-prop, and adam. For examle, lbfgs uses an approximation of the hessian as the learning rate. Interestingly for linear problems, it can be proved that using the hessian as the learning rate finds the optimal weights in one step. While more details on advanced learning methods are not given in this paper, serious deep learning researchers and competitors often use the advanced update algorithms because of the often dramatic increase in performance and decreased training time.

Normally the weights we wish to update from the cost function error are many layers deep. This example follows the model in figure 3, except with a single output instead of two. Therefore, our cost function can be redefined using the previous layer's activations, $\phi_l(\mathbf{x}) = \phi_l\left(\phi_{l-1}^T \mathbf{w}^{l-1}\right) = \phi_l\left(a^{l-1}\right)$ and $\phi_1(\mathbf{x}) = \mathbf{x}$, and a^{l-1} is considered the activation of layer $l-1$.

$$\arg\min_{\mathbf{w}} \ \text{RMSE}(\mathbf{x}) = \frac{1}{2}\sum_{N}(\mathbf{t} - \phi_4)^2 \tag{4}$$

To find the weight update for the weights in layer l in a neural network with a total number of layers L, we need to take $L - l - 1$ partial derivatives with respect to (w.r.t.) the previous inputs, and one partial derivative w.r.t. to the weights. For example, to find the weight updates for the second layer in figure 3 we take $4 - 2 - 1 = 1$ derivative w.r.t. to our previous layer's inputs, ϕ_3, and then one derivative w.r.t. to the weights of interest, $\mathbf{w}^2$.

$$\nabla^2 \text{RMSE}(\mathbf{x}) = \frac{\partial^2}{\partial \phi_3 \partial \mathbf{w}^2}\text{RMSE}(\mathbf{x}) = \sum_{N}(t - \phi_4) \times \frac{\partial \phi_3^T}{\partial \mathbf{w}^2} \tag{5}$$

$$\mathbf{w}^2_{(t+1)} = \mathbf{w}^2_{(t)} - \eta \times \nabla^2 RMSE(\mathbf{x})$$

And like in equation 3, we have our weight update for $\mathbf{w}^2_{(t+1)}$. From equation 5, we can see the equation for weight updates is the application of the chain rule. This is significant, because the chain rule will give us insight into computationally effienctly back-propogation.

$$\frac{\partial a^4}{\partial \mathbf{w}^1} = \frac{\partial a^4}{\partial a_2^3}\frac{\partial a_2^3}{\partial a_3^2}\frac{\partial a_3^2}{\partial \mathbf{w}^1} + \frac{\partial a^4}{\partial a_2^3}\frac{\partial a_2^3}{\partial a_2^2}\frac{\partial a_2^2}{\partial \mathbf{w}^1} + \frac{\partial a^4}{\partial a_2^3}\frac{\partial a_2^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial \mathbf{w}^1} + \frac{\partial a^4}{\partial a_1^3}\frac{\partial a_1^3}{\partial a_3^2}\frac{\partial a_3^2}{\partial \mathbf{w}^1} + \frac{\partial a^4}{\partial a_1^3}\frac{\partial a_1^3}{\partial a_2^2}\frac{\partial a_2^2}{\partial \mathbf{w}^1} + \frac{\partial a^4}{\partial a_1^3}\frac{\partial a_1^3}{\partial a_1^2}\frac{\partial a_1^2}{\partial \mathbf{w}^1} \tag{6}$$

Applying the chain rule to figure 4 yields equation 6. Notice how often the chain rule repeats itself. The chain rule computes two partial derivatives three times each, and computes three partial derivatives two times each. Modular back-propogation takes advantage of the redundancy so that each partial derivative needs to only be computed once.

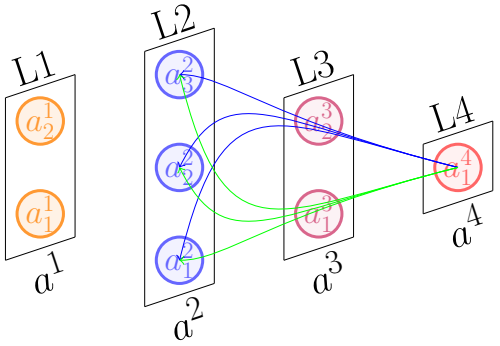

Figure 5. There are many paths error can travel through the model from the cost function output a^4. Displayed are the six paths of error required to update a single weight vector between layer 1 and 2. Notice that many of the paths go through the same two nodes, three paths for each node in layer 3. The modular view of back-propogation takes advantage of this repetition.

To view a neural network from the modular perspective, consider each function as a layer like in figure 6 for a single module, and figure 7 for an entire network. We adopt new notation for this discussion as follows. We will use z_i^l to represent the input of the i^{th} node in the l^{th} layer where there are L total layers, and z^l to represent the vectorized input for the entire l^{th} layer. We use $f_l(\cdot)$ to denote an differentiable function that outputs z^{l+1}. And we use $C = z^L$ for the output of the cost function. A single module needs to be able to perform three computations: 1) a foward pass 2) the partial derivative of previous input w.r.t. current input 3) the partial derivative of the cost w.r.t. previous layer parameters [2].

$$1)\ z^{l+1} = f_l(z^l) \tag{7}$$

$$2)\ \delta_i^l = \frac{\partial C}{\partial z_i^l} = \sum_j \frac{\partial C}{\partial z_j^{l+1}}\frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \delta_j^{l+1}\frac{\partial z_j^{l+1}}{\partial z_i^l} \tag{8}$$

$$3)\ \frac{\partial C}{\partial w^l} = \sum_j \delta_j^{l+1}\frac{\partial z_j^{l+1}}{\partial w^l} \tag{9}$$

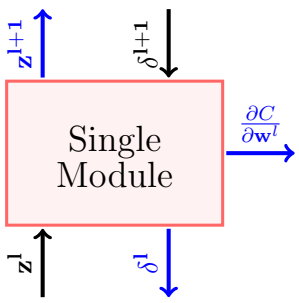

Figure 6. For full functionality of a model a single module of a neural network needs to be able to perform 3 computations, highlighted in blue.

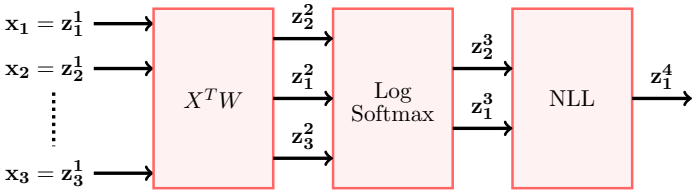

Figure 7. Pictured is a modular view of a neural network. Notice that only the first layer in this network has weights associated with it. The first layer is the linear layer. The second layer is the "log-softmax" layer. The third layer is the "negative log-likelihood" function. The second and third layers are fully explained in the logisitc regression section of this paper. If the reader is unfamiliar with their functions, it is advised that they breifly review them.

Each layer computes their respective z^{l+1} term so that it may pass it to the next layer, and the computation is done in order from input to output. For back-propogation, each layer computes the δ^l term from the output to the input, where δ^{L+1} is initialized to 1. Each module can update it's layer's weights (if it has any) whenever the complete set of previous errors, $\delta_i^{l+1} \forall i$, is recieved. Relating back to figure 5, the weight update to $\mathbf{w}^1$ can be made once layer 2 has each of their δ_i^2. This is how Torch7, a scientific package for machine learning in Lua, computes back-propogation [3]. For more details on modular back-propogation, see [2].

Throughout the discussion of the back-propogation, we have been summing from 1 through N for N training examples. This is computationally inefficient, since the entire training set must pass through the model before any weight updates can be made. This is called the *full-batch* training. In practice, what is done is *mini-batch* training. This is when the N training examples are randomly split into M subsets of N' examples. The error and back-propgation is then computed for a single M subsection of the data, and what is found is that the weight update taken is similar to the update taken when using all N examples. If $N' = 1$ then the learning is called *online-learning*.

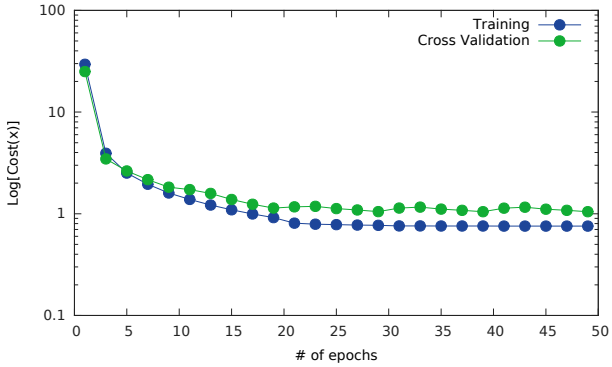

Figure 8. Full-batch versus Mini-batch versus Online learning.

A good practice is also to normalize the data either (a) across each feature, (b) across the entire

input vector, or (c) both. For instance, if you want to predict house prices you will likely have features such as number of rooms and number of square feet. We expect the square feet of a house to be much larger numbers (maybe ≈ 1000) than the number of rooms (maybe ≈ 10), so it would be wise to normalize the square feet so that the values are not magnitudes different than other features. We can also normalize the entire feature vector. For example, say we have an unrolled image as our input vector where each pixel corrosponds to a feature, and where each pixel can take values from $[1, 255]$. It would be smart to normalize the entire feature vector so that each pixel takes value from $[0, 1]$, so we divide each feature by 255. The normalization allows us to use more tools. For example if we wanted to train a network to recreate the input, we could now use the Binary Cross Entropy cost function since it's arguments must take values $[0, 1]$.

3.2 Logistic & Multinomial Logistics Regression

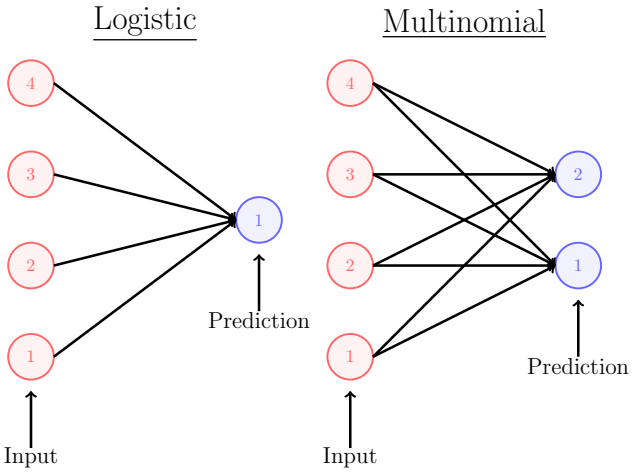

Figure 9. Logistic and multionmial logistic regression is the basic classification for binary and multi-variate classification tasks respectively. Notice how a binary task, yes or no, can be written as both a binary and multi-class problem.

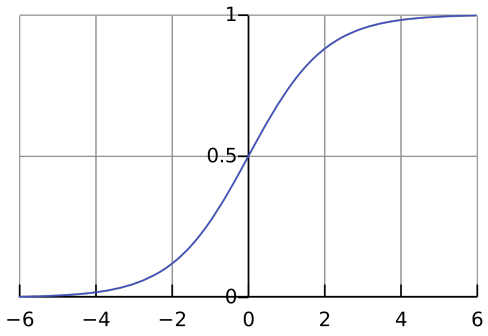

Figure 10. The sigmoid function "squashes" the input argument between $[0, 1]$, and is the fundamental building block of classification. This allows us to interpret it's output as probability. Often it is interpreted as the probability of the presence of an arbitrary feature that $\mathbf{w}$ is set to detect.

Logistic regression can be seen as the left neural network in figure 9. The first layer is an input layer consisting of input features. The second layer outputs probability that the input belongs to the class in question. The predictions are the output from the *sigmoid function*, given in equation 10 and figure 10. It squeezes the linear output between $[0, 1]$. Note this means our target vector $\mathbf{t}_i$ a single scalar value $\in \{0, 1\}$. Then it is equivalent to write $\mathbf{t}_i$ or t_i for the scalar value. This makes probabilist interpretation for binary classification straight-forward. If $\phi(\mathbf{x})$ is some basis function of $\mathbf{x}$, we get:

$$\sigma\left(\phi_i^T \mathbf{w}\right) = \frac{1}{1 + e^{-\phi_i^T \mathbf{w}}} \tag{10}$$

And formally we can say the output of the sigmoid function is expressed in the following expression.

$$p\left(C_1|\phi, \mathbf{w}\right) = \sigma;\ p\left(C_2|\phi, \mathbf{w}\right) = 1 - \sigma \tag{11}$$

The sigmoid function is interpreted as the posterior probabilty that input ϕ belongs to class C_1 or C_2. Using Baye's Rule we get:

$$\sigma(\alpha) = P(C_1|\phi) = \frac{P(\phi|C_1)\,P(C_1)}{P(\phi|C_1)\,P(C_1) + P(\phi|C_2)\,P(C_2)} \tag{12}$$

$$\phi_i^T \mathbf{w} = \ln\left(\frac{P(\phi|C_1)\,P(C_1)}{P(\phi|C_2)\,P(C_2)}\right) \tag{13}$$

Note that α depends on a linear combination of the input and parameters from the class-conditional densities. This is why logistic regression is sometimes referred to as a linear classifier despite of it's use of the sigmoid non-linear function.

Model predicitons are then fed into a cost function to compute the error. The cost function for logistic regression is the *negative-log likelihood* function (N.L.L.). This function relates to the statistical method called *maximum likelihood estimation* (M.L.E.).

Equation 14 is also known as the likelihood function of $\mathbf{t}$. Our goal is to find the set of weights $\mathbf{w}$ which maximize the likelihood function, as the name "maximum likelihood" function implies. Often it more convienent to add rather than multiply, and since the $\log(\cdot)$ function is $1:1$, it is equivalent to maximize the log(eq: 14).

$$L(\mathbf{t}, \phi, \mathbf{w}) = p(\mathbf{t}|\phi, \mathbf{w}) = p(t_1, t_2, \ldots, t_N|\phi, \mathbf{w}) = \prod_{i=1}^{N} \sigma^{t_i} (1-\sigma)^{1-t_i} \tag{14}$$

$$l(\mathbf{t}, \phi, \mathbf{w}) = \ln(p(t_1, t_2, \ldots, t_N|\phi, \mathbf{w})) = \sum_{i=1}^{N} t_i \ln(\sigma) - (1-t_i)\ln(1-\sigma) \tag{15}$$

In machine learning our goal is to minimize a cost function. Therefore, to make the interpretation of the function be more closely related to the "cost" of our model, we negate the function. Finally, the negative log-liklihood function is given by:

$$N.L.L.(\mathbf{t}, \phi, \mathbf{w}) = -l(\mathbf{t}, \phi, \mathbf{w}) = -\sum_{i=1}^{m} \ln(p(t_i|\phi, \mathbf{w})) = \sum_{i=1}^{N} t_i \ln(\sigma) - (1-t_i)\ln(1-\sigma) \tag{16}$$

The negative log-likelihood criterion is widely popular partially because of it's many interpretations. For a two class case using a single sigmoid unit, like the example above, the negative-log likelihood is also called the *binary cross-entropy* (B.C.E). The negative log-likelihood loss function measures a sudo-distance between the distibutions Y and $\hat{Y}$. More exactly, minimizing the N.L.L. is equivalent to minimizing the Kullback-Leibler Divergence [2]. We say that the N.L.L. function can be thought of as measuring the entropy, or the amount of information our input $(\mathbf{x}_i, \mathbf{t}_i)$ gives us about our model's accuracy [2]. The more "information" we recieve, the more inaccurate our model. More details on the topics of the Kullback-Leibler Divergence and entropy of a random variable are beyond the scope of this paper, however a serious machine learning student should spend time learning about them.

For multiclass cases we use multinomial logistic regression. We change our probability distribution of $\mathbf{t}$ from following a Bernoulli distribution, to following a Multinomial distribution. Therefore, instead of the sigmoid function we use the *soft-max* function. Let $z_i = \phi^T \mathbf{w}_i$, then the multinomial logistic regression neural network seen on the left in figure 9 can be written as:

$$\mathbb{I}_i\left(\mathbf{q}_j\right) = \begin{cases} 1 & \text{if } q_{j,i} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

$$\text{Soft-Max}\left(\mathbf{t}, \phi, \mathbf{w}\right) = \begin{cases} \pi_1 = \frac{e^{z_1}}{e^{z_1}+e^{z_2}} & \text{if } \mathbf{t}_i \in class\ 1 \\ \pi_2 = \frac{e^{z_2}}{e^{z_1}+e^{z_2}} & \text{if } \mathbf{t}_i \in class\ 2 \end{cases} = \pi_1^{\mathbb{I}_1(\mathbf{t}_i)} + \pi_2^{\mathbb{I}_2(\mathbf{t}_i)} = \sum_{j=1}^{j=k} \pi_j^{\mathbb{I}_j(\mathbf{t}_i)} \tag{18}$$

At the end of the equalities we see the summation form of the soft-max function, which will hold for any number of k classes, where $k = 2$ for the example equation above. The soft-max equation then redefines the multiclass version of equation 14 as:

$$L\left(\mathbf{t}, \phi, \mathbf{w}\right) = p\left(\mathbf{t}|\phi, \mathbf{w}\right) = \prod_{i=1}^{N} \sum_{j=1}^{K} \pi_j^{\mathbb{I}_j(\mathbf{t}_i)} \tag{19}$$

Again, we wish to take the log of the function so that we sum instead of multiply:

$$l\left(\mathbf{t}_i, \phi, \mathbf{w}\right) = \ln\left(p\left(\mathbf{t}|\phi, \mathbf{w}\right)\right) = \sum_{i=1}^{N} \ln\left(\sum_{j=1}^{K} \pi_j^{\mathbb{I}_j(\mathbf{t}_i)}\right) = \sum_{i=1}^{N}\sum_{j=1}^{K} \mathbb{I}_j\left(\mathbf{t}_i\right) \ln\left(\pi_j\right) \tag{20}$$

Since we defined $\mathbf{t}_i$ to be a member of only one class, the ln (sum) at the end of equation 20 simplifies to the double sum at the end of the equatlity. And the cost function can be redefined as the *log soft-max* criterion given as:

$$\text{Log Soft-max}\left(\mathbf{t}_i, \phi, \mathbf{w}\right) = -\sum_{i=1}^{N} \mathbb{I}_1\left(\mathbf{t}_i\right) \ln\left(\pi_1\right) + \mathbb{I}_2\left(\mathbf{t}_i\right) \ln\left(\pi_2\right) = -\sum_{i=1}^{N}\sum_{j=1}^{K} \mathbb{I}_j\left(\mathbf{t}_i\right) \ln\left(\pi_j\right) \tag{21}$$

Note that $\pi_k \neq 0$ since $e^z > 0$.

This is the complete formulation of logistic regression and multinomial logistic regression in a neural network framework. The weight vectors are found iteratively using back-propogation, just like in a regular neural network.

3.3 Convolutional Neural Network

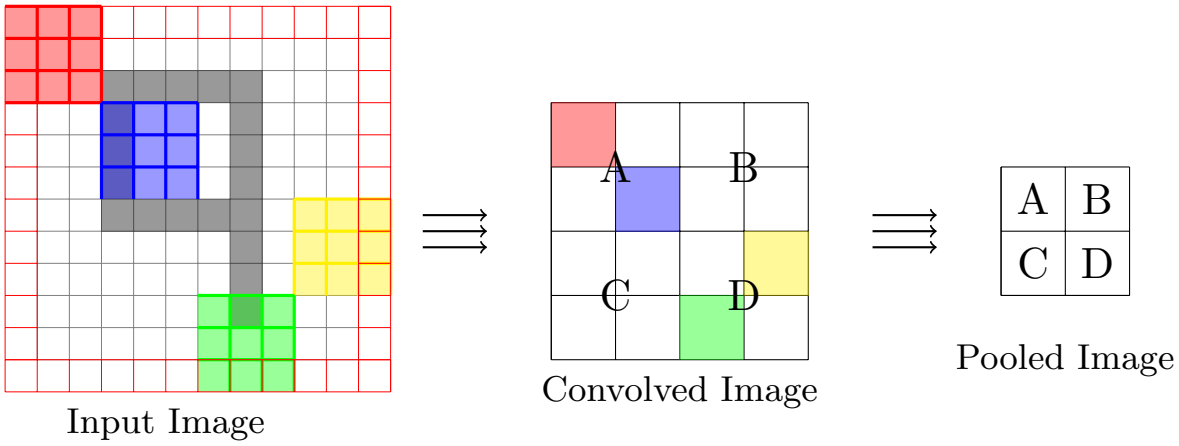

Figure 11. Each color is an instance of a single filter striding across the input image. Each of the filter indicies are multiplied with the corresponding indicies in the image, i.e. the dot product is computed. In this example, the filter has a size of 3×3 with a stride of 1 and the original input image has 1 row and 1 column of zero padding. The convolved image is then spatially pooled in 2×2 regions to create the final output from our convolution layer.

A convolutional neural network, or ConvNet, is a bona fide method for machine learning tasks including classification and natural language processing [4][5][6][7][8]. While they were popularized in the 1990's [4], convolutional nerual networks rose in popularity after Alex Krizhevsky's 2012 Imagenet competiton victory, in which his classification accuracy is 16.4%, while the next best accuracy was 26.1% [9][5]. The standard in today's networks often begin using variations of convolutional layers, so understanding this feature extraction step is an important tool for modern machine learning techniques.

For a discrete signal, which is often the case in machine learning, convoluton between an input signal $x(t)$ and a filter $h(t)$ is given by:

$$x(t) * h(t) = \sum_{\tau=-\infty}^{\infty} h(\tau)x(t-\tau) = \sum_{\tau=-\infty}^{\infty} x(\tau)h(t-\tau) \tag{22}$$

This is often thought of as a "flip-and-shift" operation. Be careful with the "flip" part of the "flip-and-shift". Intuitively it might not seem important, but without the "flip", convolution become correlation. Indeed many authors use the correlation or convolution, and while they are similar they are not the same thing. From a systems engineering perspective, convolution can be thought of as sending a signal $x(t)$ through a linear time-invariant (or more simply a "well behaved") system. Many courses devote ample classtime explaining the topic, and there are many great free online resources to learn more [10]. However, in this introductory text we will simply take the convolutional layer as feature extraction based on the assumption that nearby pixels are correlated with one another.

This assumption makes sense for problems like image classification and natural language processing. We can look at the world around us and see that context gives information for inference, and it makes sense to motivate our model structure by giving them context as well.

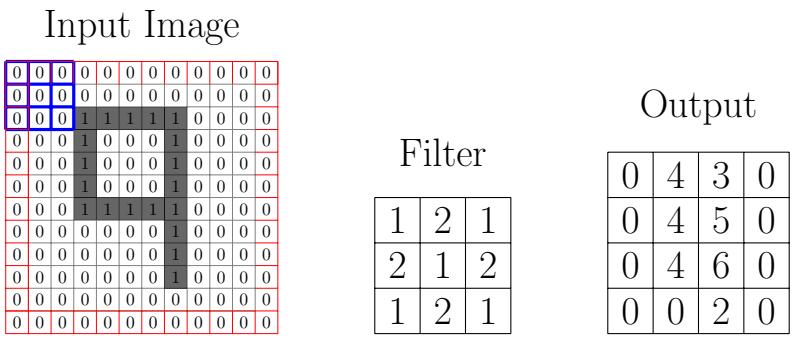

Figure 12. Visual representation of a convolutional filter over an input image. The filter is the 3×3, stride is set to 1, and there is one row and column of zero-padding which is highlighted in red. The binary digit 9 in the first column is convolved with the filter in the second column. The output of the dot product between the pixels and the filter is shown in the far right image.

The computation of a forward pass in a convolutional layer is similar to the definition given in equation 22. An input image has dimensions W×H×D, where W, H, and D is the width, height, and depth of the image. A *filter*, or kernal, can be of any size, $H_F \times W_F$, given it is smaller than the input image. The number of filters, F, gives the output of a convolutional layer it's depth. A spatial convolution refers to a 2-D filter, and a volumetric convolution refers to a 3-D filter. The filter weights and input image compute the dot product at each position along the way. Generally convolution is done in the following method. The filter starts in the corner of the input image and steps across one dimension. Once it reasches the end of one dimension, the filter increments across an orthogonal direction just once and runs along the original dimension again. This process repeats for the entire length of the orthogonal direction and for all of the orthogonal dimensions to the original dimension. The filter can skip over pixels along the convolution, and the number of pixels skipped per step is called a *stride*, S. If a filter runs along the entire image in all dimensions, the stride is set to 1. Inceasing the stride decreases the number of positions the filter visits.

After a filter and image has been convolved, the output can be thought of as an image of features. The output is always equal to or less than the size of the original image, but importantly the size of the ouput can be controlled. The output size of the convolution is a function of stride, image size, filter size, and *zero-padding*. Zero-padding, P, is an extra hyperparameter the user picks to adjust the output size of the convolution. Zero-padding adds zeros to the boarder of the input image, and the convolution runs across the new image of the resulting size. More precisely, zero-padding adds a row or column of zeros to the dimension of the image the padding is desired. In figure 12 we see paddding in 2 dimensions, H and W. We define the output size of a specific dimension, where I is the dimension of interest relating to the input image (i.e. W, H, or D). If the output of a convolution is 3-D, the depth is referred to as *channels*. The channels are the result of using multiple filters, since the filters will stack to create the depth.

$$\text{Output Dimension Size I} = \frac{(I - F + 2P)}{S} + 1 \tag{23}$$

After the output of the convolution is calculated, the convolution is generally passed through a non-linear activation function applied elementwise. A common activation function for this is the ReLU activation function in figure 13.

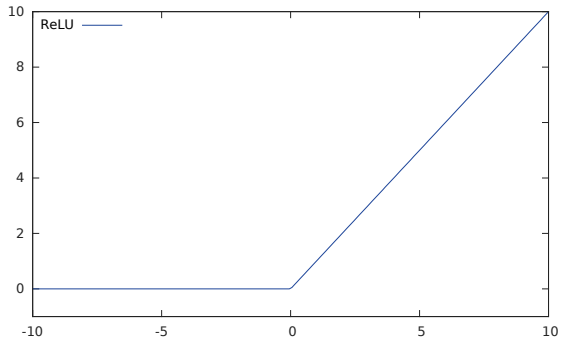

Figure 13. ReLU activation function given by $\text{ReLU}(x) = \max(0, x)$

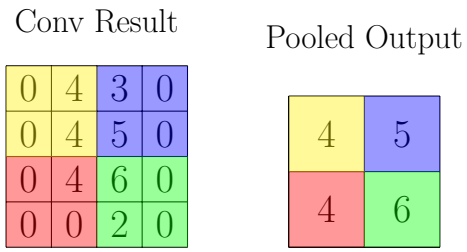

Figure 14. Max pooling of convolutional output. Each region is annotated

The final step of a convolutional layer is **pooling**. Commonly after a filter is convolved over an image, the output is sent through a non-linearity, and the final step is to "pool" the output. This step aims to shrink the resulting image down such that the most important information is retained.

In figure 14, current output from the non-linearity (ReLU) is subdivied into smaller regions. The number of regions and their degree of overlap determines the final output size. The major constraint of the regions is that they must be collectively exhaustive, meaning that every element is assigned to at least on region of pooling. Note that there is no significant improvement for overlappign regions [11] and is more computationally expensive, so it is often not done in practice.

There are two common types of pooling, *max pooling* and *average pooling*. The max pooling layer simply takes the maximum value from each region. The average pooling layer passes the average value of the region.

Now that we've computed a complete foward pass of a convolutional layer, we will now dicuss how back-propogation is used in this configuration. The pooling layers are the simplest step. The max pooling layer acts as a gate. The error simply passes through the pooling layer to the non-linearity output, and does not effect the elements that were not chosen during the forward pass. For average pooling, the error is applied by dividing it equally among the number elements in each region. The non-linearity passes error back just like an regular layer, using it's partial derivatives w.r.t. the input to propogate error backward. Given a convolution output with size H_F, W_F, F, back-propogation is computed by:

$$\frac{\partial C}{\partial w^l_{ijff`}} = \sum_{i`j`f`} \delta^{l+1}_{i`j`f`} \frac{\partial f_{i`j`f`}(\mathbf{x})}{\partial w_{ijff`}} \tag{24}$$

3.4 Support Vector Machine

Fundamentally, classification seperates datasets into different classes depending on the input features. With only 2 input features, this can be visualized by a linear decision boundarys in figure 15. We can see in the figure that there are many lines that separate the positive and negative class, but which line is the best? Visually it might seem obvious to be the black line, since that is the one that has the most even split between the closest positive and negative data points. In other words, the black line maximizes the margins between both classes.

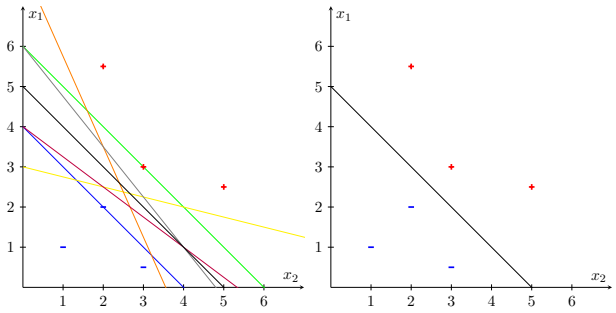

Figure 15. There are many lines which will classify with 100% accuracy, but some decision boundaries can be considered better than others. The "+" corresponds to +1 and the "-" corresponds to -1 on the third dimension not shown. The best line to separate the two classes is the line which cuts exactly between the two classes, leaving the largest margins on either side of the decision boundary (the line).

Notice how any of the lines would give us 100% accuracy, and therefore weight updates would stop once any of the lines were reach if we used a negative log-likelihood loss function. A support vector machine is meant to train a model until the "best" linear boundary is found. Given that $\mathbf{a} \equiv (a_1, \ldots, a_N)$ are the Lagrangian constraints for the N training examples of $\mathbf{x}$, the optimal decision boundary is found by using the Lagrangian function below:

$$L(\mathbf{a}) = \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} a_i a_j \mathbf{t}_i \mathbf{t}_j k\left(\mathbf{x}_i, \mathbf{x}_j\right) \tag{25}$$

For overlapping class distributions, which is frequently the case, equation 25 is subject to the constraints:

$$0 \leq a_i \leq C \tag{26}$$

$$\sum_{i=1}^{N} a_i \mathbf{t}_i = 0 \tag{27}$$

Solving for the vector $\mathbf{a}$ is a *quadratic programming* problem, and the solution can be found analytically using the SMO algorithm [12]. The SMO algorithm is nicely explained in [13]. Once the Lagrangian constraints are found, the weights $\mathbf{w}$ are calculated using following equation, where $\phi(\cdot)$ is a basis function:

$$\mathbf{w} = \sum_{i=1}^{N} a_i \mathbf{t}_i \phi\left(\mathbf{x}_i\right) \tag{28}$$

Support vector machines can be seen exactly as a neural network as in figure 17, but with a different loss function. We use a *soft-margin classifier* and implement the *hinge-loss function* given by:

$$\left[\frac{1}{n} \sum_{i=1}^{n} \max\left(0, 1 - t_i\left(\phi(\mathbf{x_i})^T \mathbf{w}\right)\right)\right] \tag{29}$$

But the above statements work only for linearly seperable datasets. If our data is not linearly separable in the original features space, like the dataset in figure 16, we use the method referred to as the *kernal-trick*. The kernal-trick is a specific kind of feature engineering to transform non-linearly separable data into a linearly seperable dataset. The end result of kernals is a new dataset $\mathbf{F} \equiv (\mathbf{f}_1, \ldots, \mathbf{f}_N)$, where $\mathbf{f}_i = (f_1, \ldots, f_N)$.

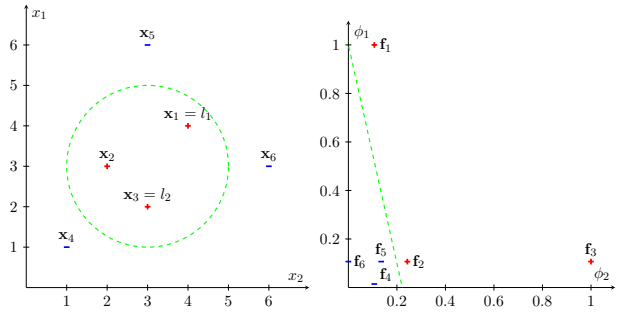

Figure 16. The original data (on the left) can not be classified by a linear decision boundary, so we must use the "kernal-trick" to transform the data set into a feature space where the data is linearly seperable (on the right). This image only shows transformation in two of the 7 dimensions (one for each training example). We used l_1 and l_3 as our "landmarks", and a guassian kernal with $\sigma^2 = 1$.

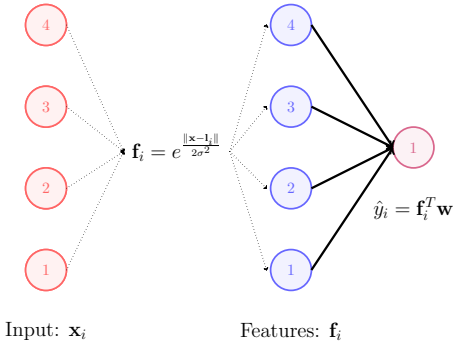

Figure 17. The kernalized support vector machine can be seen as the nerual network above, with a special activation function between the first and second layer, which requires no weights. The last layer's output is the "raw" output from a linear layer between $\mathbf{f}_i$ and $\mathbf{w}$.

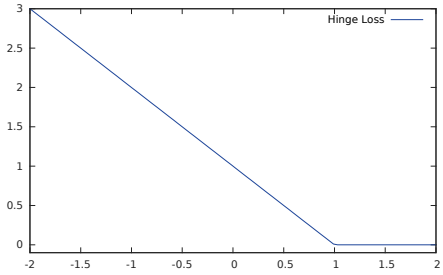

Figure 18. The Hinge-Loss function encourages "+" examples to be large in magintude and positive, and "−" examples to be large in magintude and negative.

In figure 16, each example is labeled $\mathbf{x}_i$ for the i^{th} training example. The axises are x_1 and x_2 for the 1^{st} and 2^{nd} dimension of each training example $\mathbf{x}_i$. Two of the examples, x_1 and x_3, have been selected as landmarks l_1 and l_2 respectively.

Note in figure 16 are we only using 2 landmarks for educational purposes alone. The proper procedure is to label **all** of the training examples as landmarks and contiue through the process below. We only graph the result from 2 landmarks, because we can visualize a $2-D$ plot. A $7-D$ graph is much more challenging to visualize.

For a single landmark l_i, our goal is to define a value for a single dimension of each f_i vector, $i = 1, \ldots, N$. The indicie of the new vector is arbitrary, but must remain constant for each $\{\mathbf{f}_i\}$. Generally, for N training examples there are N dimensions to define for each training example.

In figure 16, the

engineer new features from our trianing examples $\{\mathbf{x}_1\}$ into $f_1, f_2, \ldots, f_m$ for each m training examples, like in equation 3.4. The features are used as coefficients into a linear layer, and are trained on the hinge loss function. We can visualize the kernalized svm as the neural network in figure 17.

$$\text{Start by mapping every } x_i \text{ to a landmark } l_i$$
$$\text{For } i \text{ training examples}$$
$$f_{ik} = \text{similarity}(x_i, l_k) = exp(-\frac{\|l_i - x_k\|^2}{2\sigma^2}) \tag{30}$$
$$\mathbf{f}_i = (f_{i1}, f_{i2}, \ldots, f_{iN}), i = 1, \ldots, N$$
$$\text{Note: } \sigma \text{ is a user defined parameter}$$

Supoort vector machines are a powerful tool for analysis. One major benefit of support vector machine's is that their error function is convex, meaning there is a single globally best solution to the function. And even with their shallow depth, they perform well against neural networks, and are still a widely popular algorithm, which achieves test error rate of 0.56% for support vector machines compared with 0.23% for convolutional neural networks**??**.

Despite it's accuracy and popularity, support vector machines have a couple important drawbacks to consider. First is the issue of decision output, and second is the issue of multi-classification. Support vector machines output decisions, not probabilities. The label for a training point $(\mathbf{x}, \mathbf{y})$ is $y_i = 1$ if $\mathbf{x}$ belongs to the positive class, and $y_i = -1$ if $\mathbf{x}$ belongs to the negative class. Therefore, our target $\mathbf{y}$ is no longer a vector, but a single scalar value, whose values are $\{-1, 1\}$. The downside to this approach is that we no longer have a probability of our example belonging to the positive or negative class. While we can take the magnitude of the output as the confidence in our label (the furthur from zero, the more confident we are), we still lack the more accurate interpretation from probabilities. And because our output is limited to a scalar value, support vector machines are also constrained to binary classification tasks. We can break up a multi-class classification task into smaller "one-versus-all" tasks, where we consider only a single class of being positive and the other classes to be negative. That means for k classes we will have k sets of weights. Input is classified to the classifier with the largest **positive** output. These methods can lead to skewed learning, and performance suffers when the number of classes is large. If each example has the same number of $\mathbf{n}$ training examples, then when training each set of weights there are $\mathbf{n}(k-1)$ negative examples and only $\mathbf{n}$ positive examples.

3.5 Recurrent Neural Networks

While recurrent neural networks are a powerful method of supervised learning for sequence prediction, machine translation, and neural turing machines, they do not generally qualify as a classifier. This makes the discussion of recurrent networks challenging in the classificaton setting of the other supervised models. Therefore, we have decided to omit the explaination of recurrent neural networks in the paper.

4 Unsupervised Learning

4.1 K-Means Clustering

K-means clustering is a common method of unsupervised classification. The motivation is that each sample from a given class will "live" inside a similar region in a vector space. Therefore, we can classify our data without ever needing target values. The k-means clustering method is best explained by example. Consider figure 19, where we can clearly see 3 clusters of data for 3 different classes. We will use k-means clustering to classify the data.

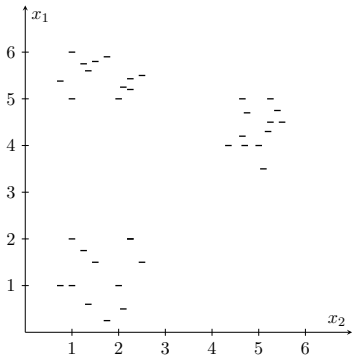

Figure 19. Pictured is the data for our k-means clustering example. There are 3 completely separate clusters. While this will give us an easy solution to our clustering problem, frequently in real world datasets class distributions overlap. In these cases, kernal-methods are used (exactly like in support vector machines) to minimize the distribution overlap.

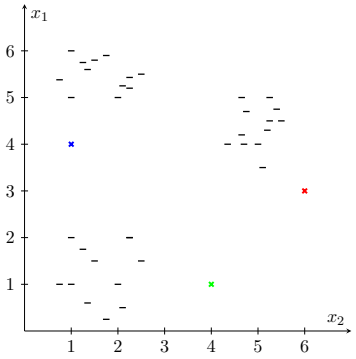

Figure 20. Here we have initialized $k = 3$ clusters. The number of centroids and their initialization is crucial for quality results.

The first step in k-means is to initialize k clusters, or centroids. In other words, we define $\mathbf{c}_i$ for $i = 1, \ldots, k$. In this example, we initialize 3 clusters as seen in figure 20. However, this is a smart guess we were able to make by simply graphing the data. The data is usually in a higher dimension, such that it is not possible to plot. Clearly, there must be a robust technique to estimate the number of clusters in the dataset. But if the technique fails, it can lead to results that have a different number of centroid than actual clusters in the data. This situation can be seen in figure 23.

TODO: So how do we find a good number of k centroids?

Ideally data points within a cluster have a small differences, relative to the difference to data points in different clusters. This can be used to check how well our data is clustered.

TODO: How do we initilized k-means?

After the number of centroids is determined and the k clusters have been initalized, we assign every training example to a cluster. This can be seen using colors in figure 21. After the assignment, the clusters are moved to the mean location of the assigned data examples. This cycle repeats until convergence.

The cycle is:

1. assign training examples
2. compute and move to the mean location

When finished, determine if any centroids are assigned to the same cluster.

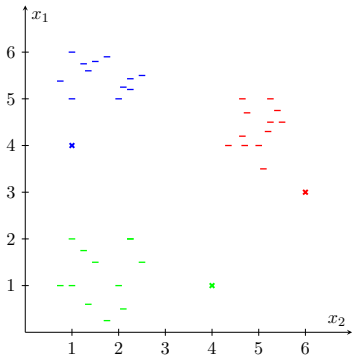

Figure 21. Every training example is assigned to the closest k cluster. The computation for distance can vary, but the Euclidean distance is common.

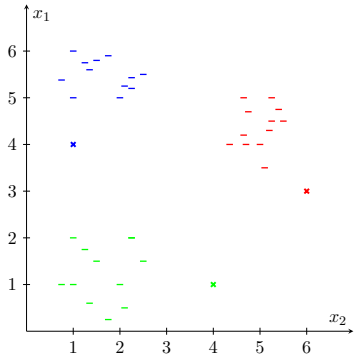

Figure 22. The k clusters move to the mean location of the assigned data-points.

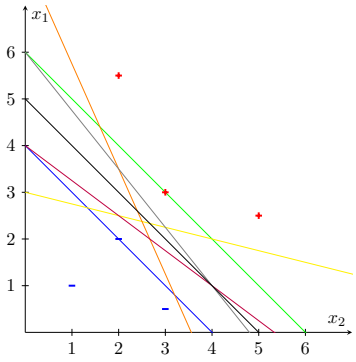

Figure 23. Are there 2 or 3 clusters? It is hard for a user to determine without looking at a graph. Computing how well the data is "clustered" can be used to determine the initial number of k means. After convergence, corrections can be made to determine if different centroid are assigned to the same cluster.

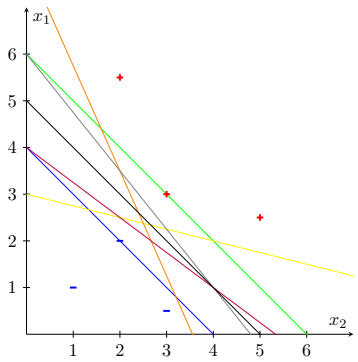

Figure 24. This is the final result of the k-means algorithm. We can clearly see that we have successfully assigned each of the training example to separate classes. Using the dataset, we can now assign the meaning to the clusters.

But clustering techniques can not solve all problems. The data must be well clustered for k-means classification to work. In fact, we can determine how well data is clustered by using the "blhasadfasdfa". For example, we can use the dispersion index of each cluster give in equation 31, where $\hat{\sigma^2}$ is the variance of the data, $\hat{\mu}$ is the mean, and $i = 1, \ldots, N$.

$$\mathrm{D}_i = \frac{\hat{\sigma^2}}{\hat{\mu}} \tag{31}$$

$$J = \sum_{i=1}^{k} \sum_{\mathbf{x} \in \mathbf{c}_i} \|\mathbf{x} - \mathbf{c}_i\|^2 \tag{32}$$

K-means custering can be seen as minimizing equation 32. The mathematical forumation of the algorithm can be seen in algorithm 1.

We can see that clustering provides a good method to classify data without using data targets. It is important to realize that k-means cannot be applied blindly to datasets, and the "clusteredness"

of the data should be considered. The importance of finding the proper number of clusters k and proper initialization is crucial to having an effective k-means algorithm.

Algorithm 1 K-Means Clustering

1: Given input that is $\mathbf{D}$ dimensional, we begin by initializing k centroids randomly in a $\mathbf{D}$ dimensional vector space.

2: $s_i, i = 1, \ldots, k$ is an empty set to contain the appropriate training examples assigned to the cluster $\mathbf{c}_i$.

3: **for** # iterations *or* until convergence **do**

4: Assignment Step: Each of the data samples $\mathbf{x}_j$, $j = 1, \ldots, N$ are assigned to the closest cluster $\mathbf{c}_i$ determined by a distance metric. In this example, we will use the Eucliean distance.

$$\mathbf{x}_j \in \{s_i : \|\mathbf{x}_j - \mathbf{c}_i\|^2 \leq \|\mathbf{x}_j - \mathbf{c}_m\|^2 \forall m, m \in \{1, \ldots, k\}\} \tag{33}$$

5: Update Step: In this step, the value of each of the data samples within each cluster assignement is averaged. The cluster is then moved to the average of the data samples. Note the $|A|$ notation denotes the cardinality of the set A, and returns the number of elements within the set.

$$\mathbf{c}_i^{t+1} = \frac{1}{|s_i^t|} \sum_{\mathbf{x} \in s_i^t} \mathbf{x}_j \tag{34}$$

6: **end**

4.2 PCA

PCA is an algorithm that uses the variance in the dataset to find an optimal linearly compression. Given data in dimension $\mathbf{D}$, PCA maps the data to dimension $\hat{\mathbf{D}}$. PCA is motivated by the face that information in the data is sometimes redundant, or useless for classification tasks. For example, consider building a classifier between a car, bike, and unicycle. Knowing that a data sample has a seat does not help us predict which class the sample belongs to. What does help is knowing about the attributes which change the most across your dataset. For instance, knowing that the training example has 4 wheels tells you that your object is a car. Reducing redundant information can greatly reduce computation costs, and more succinctly summarize your data.

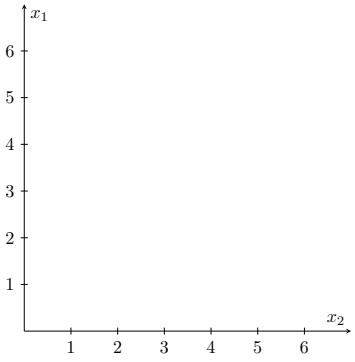

Figure 25. This is a bad direction to compress the data in, since our variability in the data is low. Notice how the red dots on the baseline are compact. Our reconstruction using the resulting red dots would be poor.

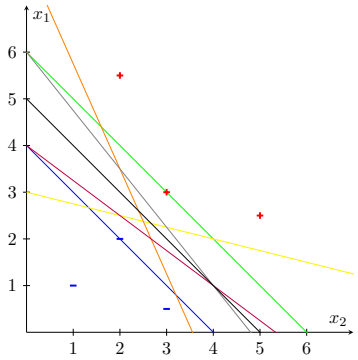

Figure 26. This is the best direction to compress the data. Clearly we have maximized the variability. Note how the line that captures the most variance also satisfies the least-squares solution for this data. Think back to high-school when you talked about the r^2 term. Linear regression is how much variance is explained by the data, which is captured in the r^2 term.

Since PCA is a set algorithm, we will explaine each step and the intution behind them. First perform mean normalization of the data. This will "center" the data around the axis. You can think of this as setting the y-intercept to zero. PCA analysis requires **two** items: 1) the direction vectors of maximum variance $\mathbf{U} \equiv \left(\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_{\hat{\mathbf{D}}}\right)$, and 2) the value of the "red" dots $\mathbf{s}_i$ as seen in figure 26.

In order to compute 1 and 2 for PCA, we must know what eigenvectors and eigenvalues are. Look at equaton 35 below. If A is some $n \times n$ matrix, then ν is a $n \times 1$ matrix call the *eigenvector* and λ is a scalar term called the *eigenvalue*. There are sometimes multiple eigenvectors and eigenvalues for a given matrix A.

$$\mathrm{A}\nu = \lambda\nu \tag{35}$$

After mean normalizing the data, we must compute the eigenvalues and eigenvectors for the covariance matrix of the data. The covariance matrix is defined in equation 36. The direction of greatest variance is the eigenvector with the largest corresponding eigenvalue. Therefore the vectors u_i are given by thes eigenvectors ν. This satifies step 1. To compress the data to dimension $\hat{\mathbf{D}}$, we keep the eigenvectors ν_l correpsonding to the largest eigenvalues, λ_l, for each dimension in $\hat{\mathbf{D}}$. This means we through elimenate the dimensions with low variance, and the vectors $\mathbf{u}_i \in \mathbf{U}$ are the eigenvectors pointing in the direction of highest variance.

$$\Sigma(\mathbf{X}) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i^T \mathbf{x}_i = \begin{bmatrix} \mathrm{var}(x_1) & \mathrm{cov}(x_1, x_2) & \ldots & \mathrm{cov}(x_1, x_n) \\ \mathrm{cov}(x_2, x_1) & \mathrm{var}(x_2) & \ldots & \mathrm{cov}(x_2, x_n) \\ \mathrm{cov}(x_n, x_1) & \mathrm{cov}(x_n, x_2) & \ldots & \mathrm{var}(x_n, x_n) \end{bmatrix} \tag{36}$$

To compute step 2, calculating the $\mathbf{s}_i$, we need to calculate the projection of the original data onto the u_i pointing in the direction of maximum variance. The formula is given in equation 37. Remeber that $\mathbf{U}^T$ is a $\hat{\mathbf{D}} \times \mathbf{D}$ matrix, and $\mathbf{x}_i$ is a $\mathbf{D} \times 1$ vector. Therefore the final output is a $\hat{\mathbf{D}} \times 1$ vector.

$$\mathbf{s}_i = \mathbf{U}^T \mathbf{x}_i \tag{37}$$

PCA is an excellent linear compression algorithm. However, linear systems are limited in power. In the next section we will look at auto-encoders, which is a non-linear method of data compression.

4.3 Auto-Encoders

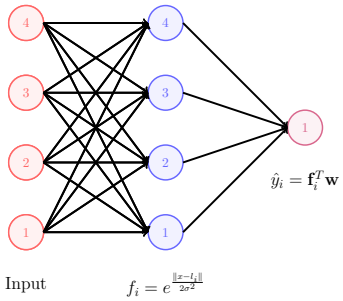

Figure 27. This figure represent the architecture of an auto-encoder. Auto-encoders are simply a special case of neural networks. They create low-dimensional, non-linear representations of data.

Auto-encoders provide a non-linear method of data compression.

5 Hardware and Platform

5.1 GPU Performance

The computationally intensive iterative training of process is a limiting factor in creating completely trained machine learning models. There are two popular solutions to this constraint. One is to use *distributed computing*. The motivation behind distributed computing is to spread the computation among numerous computers, or a computer cluster. This emulates a powerful computer, where no one computer is powerful by themselves. Computer clusters require specialized software, such as Apache Hadoop, to manage distrubtion of data. Running machine learning processes on across a computer cluster requires more software, like the Apache Spark API.

The other method to deal with the computational intensity of training is utilizing a *Graphical Processor Unit* or *GPU*. GPU's are a standard for any serious deep learning research. Alex Krizhevsky's 2012 milestone work on ImageNet utilized the power of parallel GPU's for training [5]. Today NVIDIA creates many popular deep-learning GPU's including the current reigning non-commercial GPU champion, the GeForce GTX GeTitan X pictured in figure 28. For more information on GPU's, visit my collegue's webpage [14].

Figure 28. The NVIDIA GeForce GTX Titan X is a powerful non-commercial GPU. This GPU boasts 3072 Cuda cores, 1000 MHz base clock, 1075 boost clock, 7 Gbps memory clock, 12 GB of GDDR5, and 336.5 GB/sec memory bandwidth. The retail value of a single Titan X is around $1000, and often deep learning researchers and competitors will purchase a computer with 4 of them.

5.2 Coding Platforms

There are many option when deciding to code for a machine learning platform. Included are Theono, Tensor Flow, Caffee, Torch7, and many more. Nando de Freitas' Oxford University online courses incorperated Torch7, and there were 6 practicals using the assignment [2]. Therefore, I used Torch7 for each of my models [3]. My code for the project is available online at [15].

6 Data Analysis: MNIST

6.1 Classification Results

Model	Training Error	Testing Error	CV Error	# Misclassified
CNN	99.99%	98.74%	98.69%	307
MLP	99.12%	96.74%	96.47%	1119
SVM	99.00%	96.69%	96.29%	1202
Logit	91.25%	88.79%	88.13%	6683

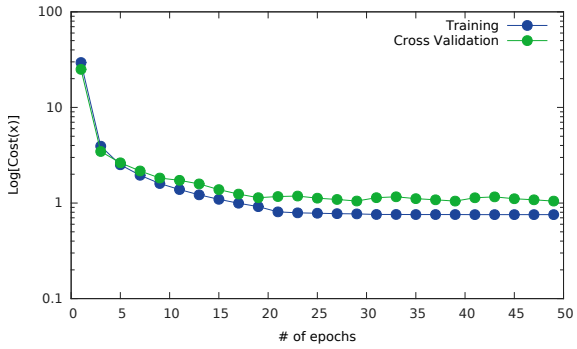

Figure 29. Fitness of Multi-Layer Perceptron.

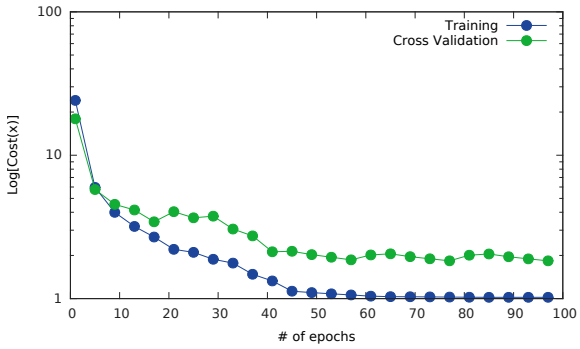

Figure 30. Fitness of Logistic Regression

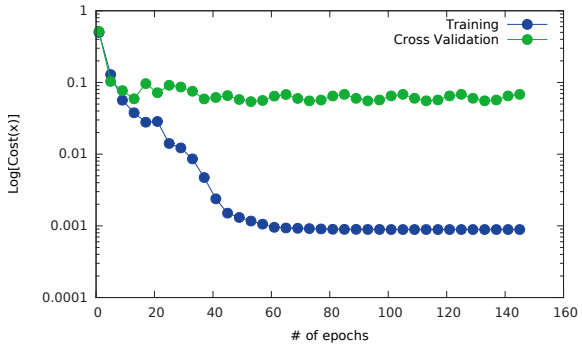

Figure 31. Fitness of Convolutional Neural Network

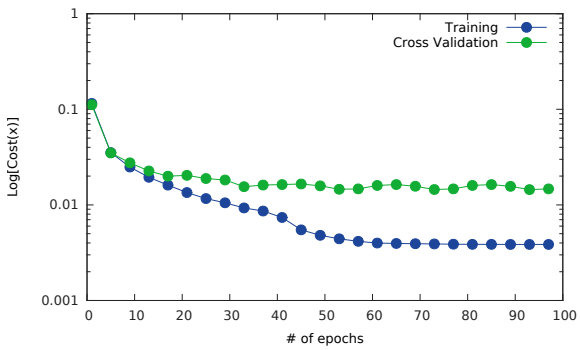

Figure 32. Fitness of Support Vector Machine [1]

[1] Loss computed using Hinge-Loss function, instead of NLL like the other 3 models. Therefore, direction comparision is not appropriate. Instead notice at the convergence, and the differenes between training and cross validation errors.

6.2 Features Visualization

Figure 33. After clustering we can color each cluster to produces an image like this. We can interpret the different colors as the regions that separate the classes.

Figure 34. A compressed representaion of the mnist digits using PCA. **more details on the compression**

Figure 35. A compressed representaion of the mnist digits using auto-encoders. **more details on the compression**

Figure 36. The results on the right have the smallest Euclidean distance between the compressed representations of the image on the left and itself. This means that we can interpret the compressed representation as having some correlation to the content of the image.

7 Conclusion

References

[1] Yann LeCun. *The MNIST Database*. 1998. URL: `http://yann.lecun.com/exdb/mnist/index.html`.

[2] Nando de Freitas. *Oxford University: Machine Learning Course*. 2015. URL: `https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/`.

[3] R. Collobert, K. Kavukcuoglu, and C. Farabet. "Torch7: A Matlab-like Environment for Machine Learning". In: *BigLearn, NIPS Workshop*. 2011.

[4] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[6] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). URL: `http://arxiv.org/abs/1512.03385`.

[7] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014).

[8] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. "A Convolutional Neural Network for Modelling Sentences". In: *CoRR* abs/1404.2188 (2014). URL: `http://arxiv.org/abs/1404.2188`.

[9] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[10] Steve Crutchfield. *Joy of Convolution*. 2016. URL: http://pages.jh.edu/~signals/convolve/.

[11] Dominik Scherer, Andreas Müller, and Sven Behnke. "Evaluation of pooling operations in convolutional architectures for object recognition". In: *International Conference on Artificial Neural Networks*. Springer. 2010, pp. 92–101.

[12] John C. Platt. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Tech. rep. ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING, 1998.

[13] Andrew Ng. "CS229 Lecture Notes: Support Vector Machines". In: 2012.

[14] Karan Samel. *The eGPU Experience*. 2016. URL: http://karans.github.io/iPinYouDNN/.

[15] Kent Gauen. *Torch7 Machine Learning Models*. 2016. URL: http://llc.stat.purdue.edu/~gauenk/.