From Pixels to Predictions: A Comparative Study of CNNs and SVMs in Image Classification

Aditi Raju, Alan Liu, Jimi Abbott, Masha Zaitsev

December 11th, 2023

This project aims to compare Convolutional Neural Networks (CNNs) and Support Vector Machines (SVMs) in image classification. Focusing on accuracy, efficiency, and scalability, we will test both methods on various datasets and explore them mathematically. The goal is to provide clear benchmarks for their performance, helping to guide the choice of classification techniques in diverse applications within the field of machine learning.

Contents

1 An Exploration of Support Vector Machines **2**
- 1.1 Maximal Margin Classifiers . . . 3
  - 1.1.1 Intuition . . . 3
  - 1.1.2 Maximal Margin Classifiers Mathematically . . . 3
- 1.2 Support Vector Classifiers . . . 4
  - 1.2.1 Intuition . . . 4
  - 1.2.2 Support Vector Classifiers Mathematically . . . 4
  - 1.2.3 Simulated Data Example . . . 5
- 1.3 Support Vector Machines . . . 10
  - 1.3.1 Intuition . . . 10
  - 1.3.2 Support Vector Machines Mathematically . . . 10
  - 1.3.3 SVMs for More than 2 Classes . . . 11
  - 1.3.4 Simulated Data Example . . . 12
- 1.4 Conclusion . . . 16

2 Experimenting with CIFAR-10 and Support Vector Machines **17**
- 2.1 Loading CIFAR-10 Data . . . 18
- 2.2 Experimentation with Different Kernels . . . 20
- 2.3 Tuning Hyperparameters (C) . . . 21
- 2.4 Evaluating the Model using Dimensionality Reduction . . . 23
- 2.5 Multi-class Classification with SVMs . . . 26

2.6 Final Results and Conclusions 27

3 An Exploration of Convolutional Neural Networks 28
3.1 Basis of Neural Networks 28
3.2 Convolutional Neural Networks 29
3.3 Fitting a CNN 32
3.3.1 Feedforward 32
3.3.2 Backpropogation 33
3.4 Simulated Data Example 34

4 Experimenting with Convolutional Neural Networks Using CIFAR-10 38
4.1 Imports 39
4.2 Loading the Dataset 39
4.3 Model 40
4.4 Training the Model 42
4.5 Testing the Model 45
4.6 Hyperparameter Tuning 46
4.6.1 Batch Size: Training Time vs. Performance 46
4.6.2 Number of Epochs 47
4.6.3 Learning Rate 48
4.6.4 Optimizers: Momentum SGD, Adam, RMSProp 48

5 SVM and CNN Comparison on Real-World Data 50

6 Contributions 51

7 Sources 51

1 An Exploration of Support Vector Machines

Support Vector Machines (SVMs) are a type of supervised machine learning algorithm primarily used for classification tasks. The main idea behind SVM is to find a hyperplane in a multidimensional space that distinctly classifies the data points. To achieve this, SVM looks for the hyperplane that has the largest margin, meaning the greatest distance between the hyperplane and the nearest data point from each class. This is achieved by leveraging support vectors, which are the data points closest to the hyperplane. These support vectors are critical in defining the position and orientation of the hyperplane. The algorithm is versatile, as it can handle both linear and non-linear separations by using different types of kernel functions to transform the data into a higher dimension where a linear separator can be found. SVMs are known for their robustness, especially in high-dimensional spaces, and their ability to model complex boundaries with a relatively simple mathematical framework. For this section, "An Introduction to Statistical Learning" [1] is used as a reference.

1.1 Maximal Margin Classifiers

Before diving into support vector machines, it's important that we build our intuition by discussing the maximal margin classifier and the support vector classifier. These two classifiers will lead us nicely into SVMs.

1.1.1 Intuition

The idea of the maximal margin classifier is that we want to find the hyperplane that perfectly separates the classes of data from each other. First, we define our hyperplane as

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p = 0$$

We say that our hyperplane is divided into two-halves; that is, for a given X, if the equation computes to less than 0, we say that is one one side of the hyperplane, and if it computes to greater than 0, we say that it is on the other side. So, in using this method, we assume that the boundary that separates our data is linear. We also assume that there exists a hyperplane that separates the data perfectly, which is why we don't actually use this method in practice.

When our data can be separated perfectly, there are an infinite amount of hyperplanes that we can use to separate our data. So, when choosing our hyperplane, we choose the hyperplane that is farthest from our training observations. So, we have a certain region in which no training observations exist, and we choose the hyperplane that is in the middle of the region.

1.1.2 Maximal Margin Classifiers Mathematically

Now, we will show how we obtain the maximal margin classifier for a set of data mathematically. Recall the idea of the method: we want to find the hyperplane that separates the classes of data and exists perfectly in between the classes of data.

Let M be the margin of our hyperplane. We want to maximize M with respect to our hyperplane. Let there be n training observations, where $x_1, ..., x_n \in \mathbb{R}^p$, where p is the number of predictors. Let our class labels be defined as $y_1, ..., y_2 \in \{-1, 1\}$. This is an optimization problem: $\max_{\beta_0, \beta_1, ..., \beta_p, M} M$

with respect to

$$\sum_{j=1}^{p} \beta_j^2 = 1$$

where

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}) \geq M, \forall i = 1, ..., n$$

Recall that this formulation is only valid when there exists a hyperplane that can perfectly separate the data. Because this is not practical with real data, we will now consider methods that have "soft" margins that allow for some overlap.

1.2 Support Vector Classifiers

The maximal margin classifier is important in laying the groundwork for an important set of methods in machine learning, but it cannot be used in practice due to its constraints. So, we introduce the support vector classifier, which involves a tuning parameter that allows for overlap between classes of data.

1.2.1 Intuition

Instead of finding the largest possible margin for our perfectly separated data, the support vector classifier allows some observations to be on the wrong side of the margin and/or the wrong side of the hyperplane. It still assumes the data is separable by a linear boundary. Just like any other machine learning method, the support vector classifier introduces a bias-variance trade-off with its tuning parameter, C. When C is relatively small, our margin is large, and so the classifier is less fit to the data, leading to higher bias and lower variance. When C is relatively large, our margin is small, and so the classifier is less biased but has more variance. Interestingly, in our mathematical exploration of this method, we learn that the only data points that affect the formulation of the classifier are those that lie on the margin or on the wrong side of the margin.

1.2.2 Support Vector Classifiers Mathematically

Similar to the formulation of the maximal margin classifier, the formulation of the support vector classifier is an optimization problem with a few more details involved.

Let $C > 0$ be our tuning parameter. Let $\epsilon_1, ..., \epsilon_n$ be "slack" variables that allow individual observations to be on the wrong side of the hyperplane or margin. C bounds these. The support vector classifier is the solution to the optimization problem: $\max_{\beta_0,\beta_1,...,\beta_p,\epsilon_1,...,\epsilon_n,M} M$

subject to

$$\sum_{j=1}^{p} \beta_j^2 = 1,$$

where

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}) \geq M(1 - \epsilon_i)$$

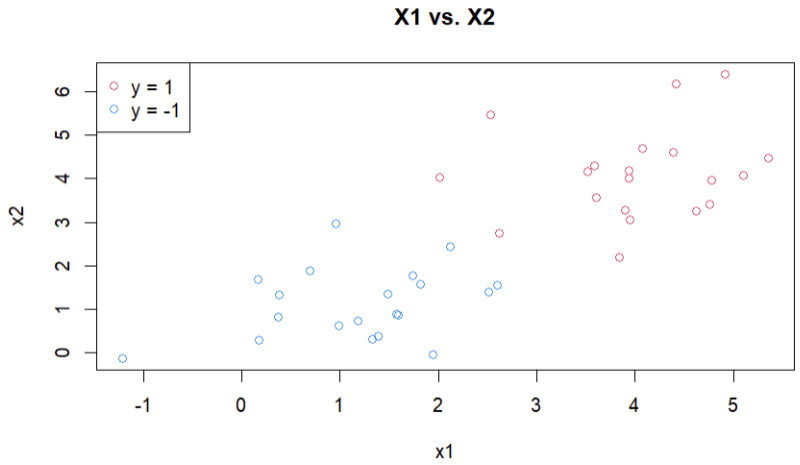
$$\epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \leq C$$

If $\epsilon_i = 0$ then the ith observation is on the correct side of the margin. If $\epsilon_i > 0$ then the ith observation is on the wrong side of the margin. If $\epsilon_i > 1$ then the ith observation is on the wrong side of the hyperplane.

1.2.3 Simulated Data Example

We'll show how support vector classifiers work using a couple different simulated data examples, with the goal of showing how they perform on different data and how the tuning parameter C works. To use SVCs in R, we need the e1071 library and the svm function where we set kernel = "linear". For these examples we'll be using two predictor variables so we can easily visualize results.

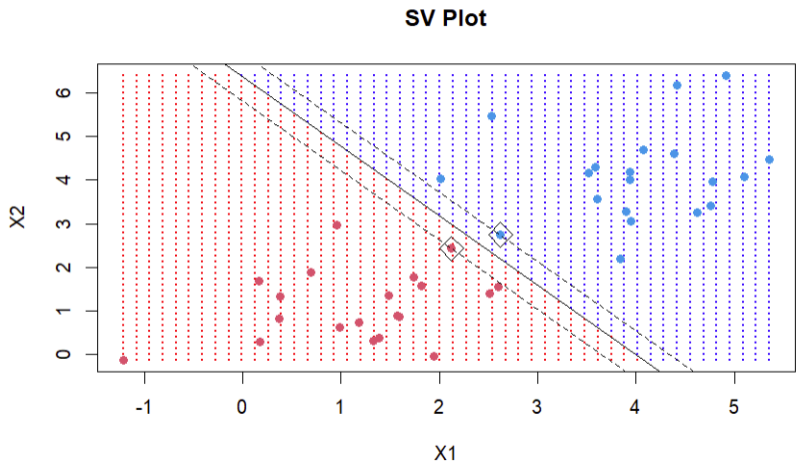
```
library(e1071)

set.seed(1)
x1 <- rnorm(40, mean = 1)
x2 <- rnorm(40, mean = 1)
X <- matrix(c(x1, x2), ncol = 2)
y <- c(rep(-1, 20), rep(1, 20))
X[y == 1, ] <- X[y == 1, ] + 3
data = data.frame(X, y = as.factor(y))
plot(X, col = (3 - y), xlab = "x1", ylab = "x2")
legend("topleft", legend = c("y = 1","y = -1"), col = c(2, 4), pch
↪  = c(1, 1))
```



Take a look at this plot: the red points, where y = 1, are linearly separable from the blue points, where y = -1. Hopefully, looking at this plot, we can visualize the idea of how we'd expect our support vector classifier to look like. Using base plotting and the svm model to plot is a little weird, so we'll take a different approach to create our own plot that visualizes the SVC nicely.

```
svm.fit <- svm(y ~ ., data = data, kernel = "linear", cost = 10,
↪  scale = FALSE)

```

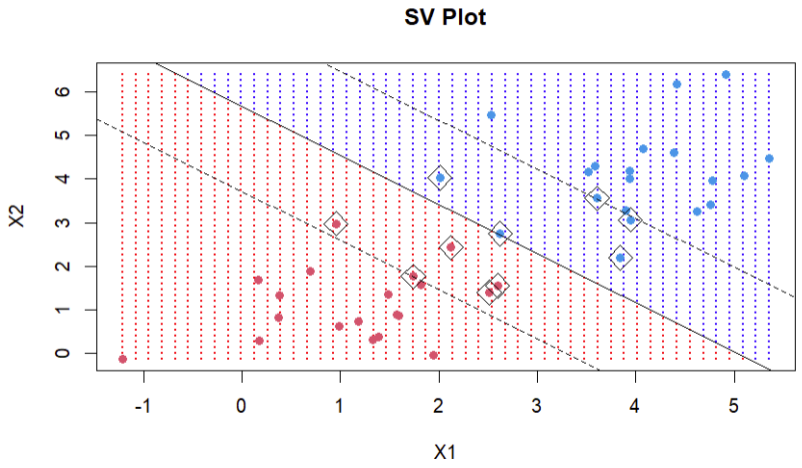
```r
svm_plot <- function(X, y, svm.fit) {
  grange = apply(X, 2, range)
  x1 = seq(from = grange[1,1], to = grange[2,1], length = 50)
  x2 = seq(from = grange[1,2], to = grange[2,2], length = 50)
  xgrid = expand.grid(X1 = x1, X2 = x2)
  ygrid = predict(svm.fit, xgrid)
  beta = drop(t(svm.fit$coefs)%*%X[svm.fit$index,])
  beta0 = svm.fit$rho
  plot(xgrid, col = c("red", "blue")[as.numeric(ygrid)], pch = 20,
  ↪  cex = .2, main = "SV Plot")
  points(X, col = y + 3, pch = 19)
  points(X[svm.fit\$index,], pch = 5, cex = 2)
  abline(beta0 / beta[2], -beta[1] / beta[2])
  abline((beta0 - 1) / beta[2], -beta[1] / beta[2], lty = 2)
  abline((beta0 + 1) / beta[2], -beta[1] / beta[2], lty = 2)
}

svm_plot(X, y, svm.fit)
```



This is a great visualization for the result of the fit our support vector classifier. The solid line represents the hyperplane, while the dotted lines represent the margin. In this example, we used a cost of 10 for our support vector classifier. This seems to be a relatively large C for our data as our support vector classifier looks like we'd expect the maximal margin classifier to look like; that is, the only points on the margin are the two that determine the margin. Let's try some different values and see if that changes the formulation of the SVC.

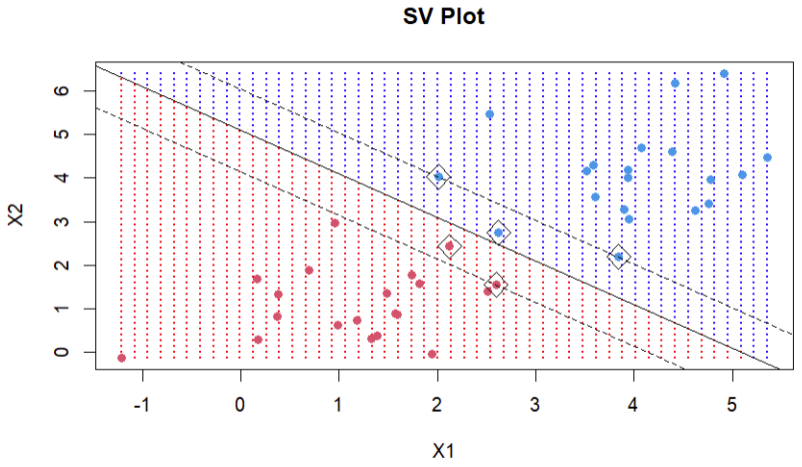
```r
svm.fit <- svm(y ~ ., data = data, kernel = "linear", cost = 0.1,
↪  scale = FALSE)
```

```
svm_plot(X, y, svm.fit)
```



A cost of 0.1 leads to a larger margin, as we expected, and a different hyperplane.

```
svm.fit <- svm(y ~ ., data = data, kernel = "linear", cost = 1,
↪  scale = FALSE)
svm_plot(X, y, svm.fit)
```

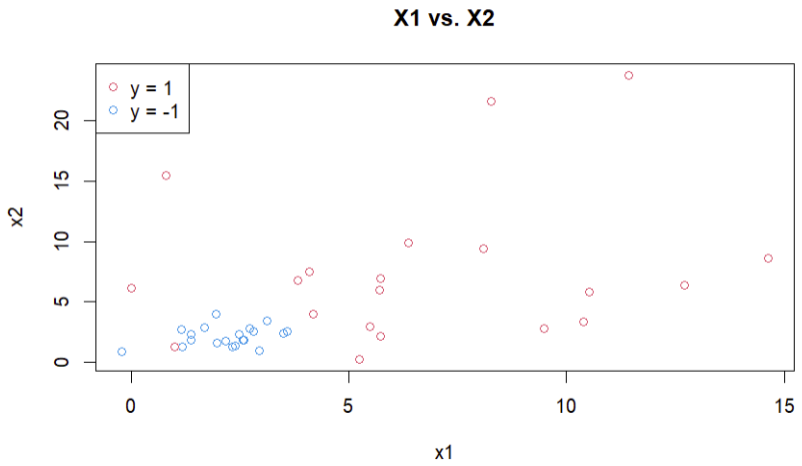


A cost of 1 leads to a support vector classifier with 5 total points in or on the margin, compared to 2 from our first rendition.

Ultimately, when working with the tuning parameter, C, it's not easy to determine a good C before running our model, so don't be afraid to mess with different values to find a model that finds a good balance between bias and variance.

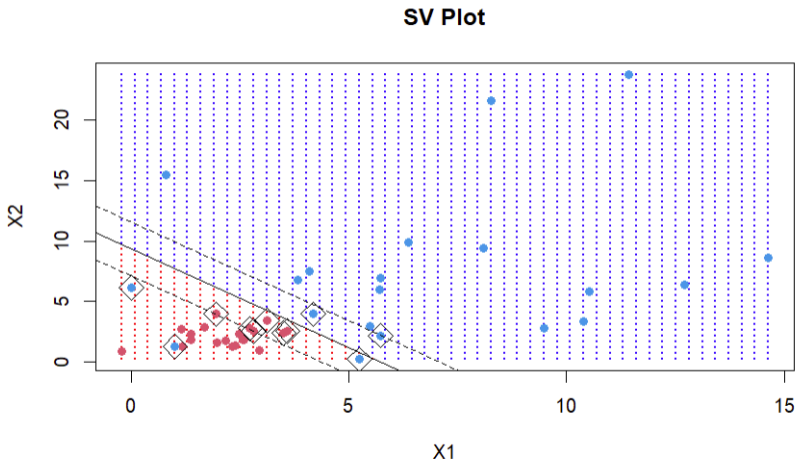
Now, let's alter our original dataset to take a look at how the support vector classifier performs on data that is not as linearly separable.

```
set.seed(1)
x1 <- rnorm(40, mean = 2)
x2 <- rnorm(40, mean = 2)
X <- matrix(c(x1, x2), ncol = 2)
y <- c(rep(-1, 20), rep(1, 20))
X[y == 1, ] <- X[y == 1, ]**2 + X[y == 1,]
plot(X, col = (3 - y), xlab = "x1", ylab = "x2")
legend("topleft", legend = c("y = 1","y = -1"), col = c(2, 4), pch
↪  = c(1, 1))
data = data.frame(X, y = as.factor(y))
```



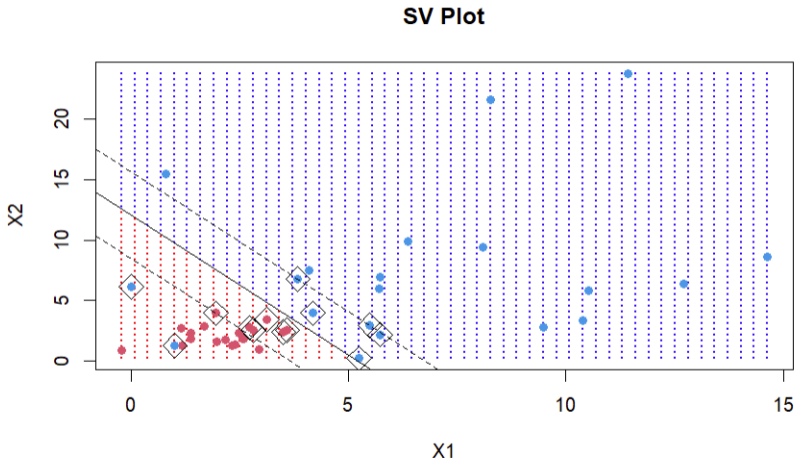
As we can see from this plot, we have data that is pretty well separated from each other. However, it's less clear how a linear boundary would look on this data. Let's take a look at some examples with some different C values.

```
svm.fit <- svm(y ~ ., data = data, kernel = "linear", cost = 1000,
↪  scale = FALSE)
svm_plot(X, y, svm.fit)
```



Looking at our original plot, it seems like we could reasonably draw a boundary that could classify all but 1 point correctly. However, we are not able to achieve that success using a support vector classifier with a high C value. We'll use a lower C value to achieve a lower variance model.

```
svm.fit <- svm(y ~ ., data = data, kernel = "linear", cost = 1,
↪  scale = FALSE)
svm_plot(X, y, svm.fit)
```



In this case, our classifier with a much lower C value actually looks similar on the training data compared to the high C value model, showing that our data is less linearly separable and a lower variance approach is more reasonable.

What if we wanted to use a non-linear boundary to separate this data? Well, we use SVMs!

1.3 Support Vector Machines

Support vector machines are very similar to support vector classifiers except they are able to fit non-linear data. This makes working with different C values more complex, but this added flexibility allows us to use SVMs in many different cases.

1.3.1 Intuition

Support vector machines are an extension of the support vector classifier by way of enlarging the feature space using kernels. So, the SVM is linear in this higher dimension, but in our original feature space, our boundary takes a non-linear form. This allows for more computational efficiency in contrast to actually using our higher dimension feature space. For SVCs, we strictly use a linear kernel, as we saw in our code from earlier. For SVMs, we use different types of kernels, Conceptually, the polynomial version of the SVM amounts to fitting a support vector classifier in a higher dimensional space using polynomials of degree d. When d = 1, the SVM reduces to the support vector classifier. Another type of kernel we can use is the radial kernel. Instead of fitting data using polynomial transformations of X, we formulate our SVM using Euclidean distances.

1.3.2 Support Vector Machines Mathematically

The two most common kernels outside of the linear (SVC) kernel are the polynomial and the radial kernels. We will first take a look at the SVM using the linear kernel (SVC), which will lead us nicely into the polynomial .

The calculation of the SVC using the kernel trick involves taking the inner products of observations of our data. The inner product of two observations, $x_i, x_{i'}$, is defined as

$$\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j}$$

We define the support vector classifier as

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle$$

β_0 and $\alpha_1, ..., \alpha_n$ are our parameters, where each α_i corresponds to the ith observation. So, each training observation has a parameter. We estimate these parameters by calculating the inner products between every pair of observations. However, only observations that are support vectors have a non-zero α_i, so we

can rewrite our function to reduce our calculations. Let S be the collection of indices of our support points. Now, we have

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i \langle x, x_i \rangle$$

Let $K(x_i, x_{i'})$ be our kernel, where

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j}$$

for our linear kernel. Now, our SVC takes the form

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x, x_i)$$

We can generalize our classifier to non-linear boundaries just by changing our kernel function. For example, if we'd like to use a polynomial kernel, we let

$$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d$$

where d is our degree. When d = 1, our classifier reduces to the SVC. The kernel function is very powerful, allowing us to easily mathematically switch between different methods of boundary formations. If the boundary between our classes isn't linear or can't be easily represented using polynomials, we can also use a radial kernel. In this case, we have

$$K(x_i, x_{i'} = \exp(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2)$$

where γ is a positive constant. The idea of the radial kernel is the following: we use the Euclidean distance to determine the locality of an observation, so only nearby training observations have an effect on the classification of a test observation.

Ultimately, the effectiveness of the kernel is that we can expand the complexity of our classifier without having to work in a higher dimensional feature space, which is computationally advantageous for us.

1.3.3 SVMs for More than 2 Classes

A single support vector machine does not support more than two classes. However, we have two methods that work around this problem: the One-Versus-One Classification, and the One-Versus-All Classification.

The One-Versus-One Classification method for SVMs fits $\binom{K}{2}$ SVMs, where K is the number of classes. Each of these SVMs compares a unique pair of classes.

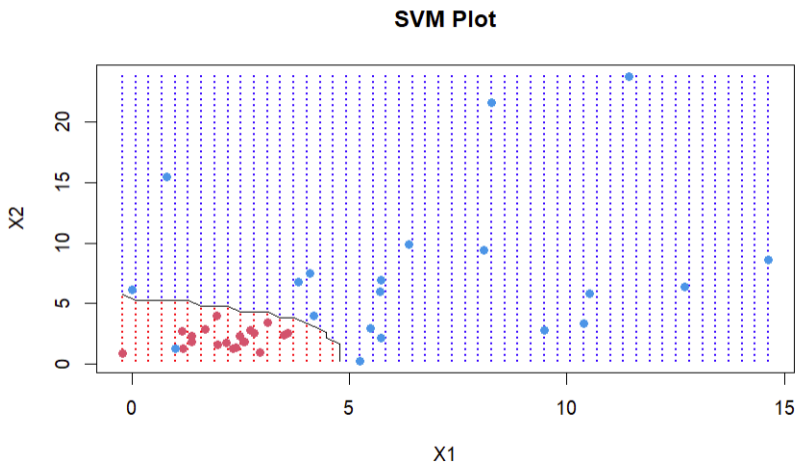
We classify a test observation using each of the SVMs, and tally the number of times it is assigned to each of the K classes. Then, we assign the observation to the class with the most tallies. This is an intuitive and thorough method, but may become computationally inefficient as K gets larger.

The One-Versus-All Classification method for SVMs fits K SVMs, where K is the number of classes. Each of the SVMs compares one of the classes to the remaining K - 1 classes. To explain how we classify a test observation, we first let $\beta_{0k}, \beta_{1k}, ..., \beta pk$ denote the parameters that result from fitting an SVM comparing the kth class, coded as $+1$, to the other classes, coded as -1. We let x^* be a test observation, and we assign x^* to the class for which $\beta_{0k} + \beta_{1k}x_1^* + \cdots + \beta_{pk}x_p^*$ is largest.

1.3.4 Simulated Data Example

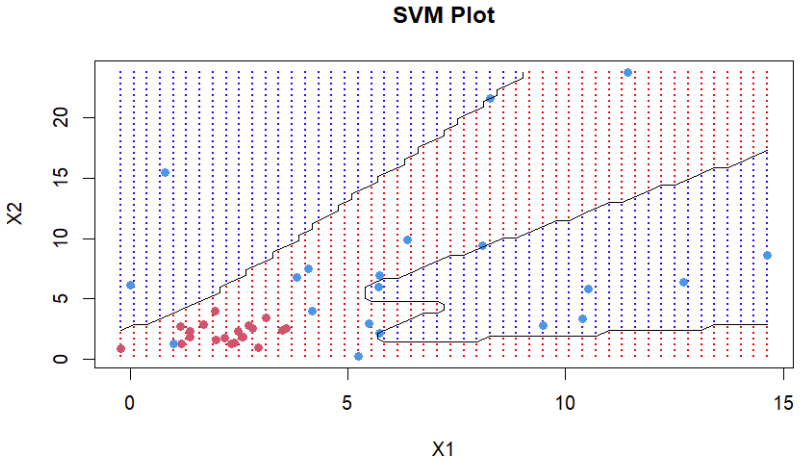
Recall our previous example where we fit an SVC on data that had a not-so linear boundary. We will reuse that data for this example and see how the SVM with a polynomial and radial kernel looks on the data.

```r
svm_plot.k <- function(X, y, svm.fit) {
  grange = apply(X, 2, range)
  px1 = seq(from = grange[1,1], to = grange[2,1], length = 50)
  px2 = seq(from = grange[1,2], to = grange[2,2], length = 50)
  xgrid = expand.grid(X1 = px1, X2 = px2)
  ygrid = predict(svm.fit, xgrid)
  plot(xgrid, col = c("red", "blue")[as.numeric(ygrid)], pch = 20,
  ↪  cex = .2, main = "SVM Plot")
  points(X, col = y + 3, pch = 19)
  decision.values = predict(svm.fit, xgrid, decision.values = TRUE)
  decision.matrix = matrix(decision.values, length(px1),
  ↪  length(px2))
  contour(px1, px2, decision.matrix, levels = c(-1, 0, 1), add =
  ↪  TRUE)
}

svm.fit <- svm(y ~ ., data = data, degree = 3, kernel =
↪  "polynomial", cost = 10, scale = FALSE)
svm_plot.k(X, y, svm.fit)
```



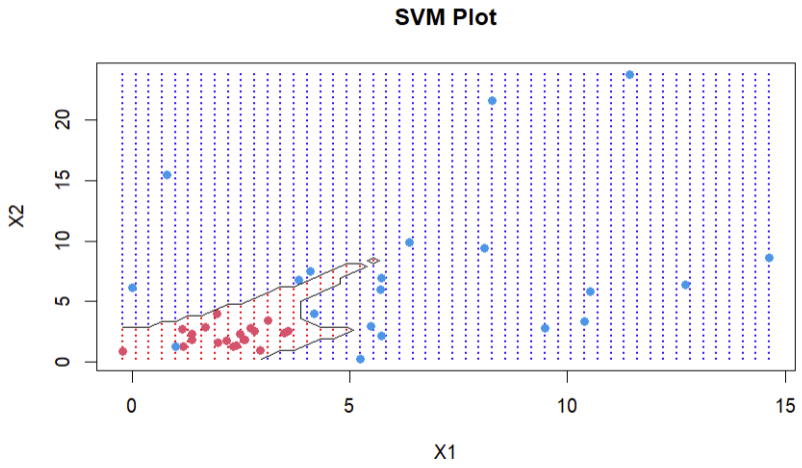
With a SVM with degree 3 and C = 10, we see a boundary that fits around the data pretty nicely without overfitting. Let's try a different degree of polynomial with the same cost to see how that changes.

```
svm.fit <- svm(y ~ ., data = data, degree = 5, kernel =
↪  "polynomial", cost = 10, scale = FALSE)
svm_plot.k(X, y, svm.fit)
```



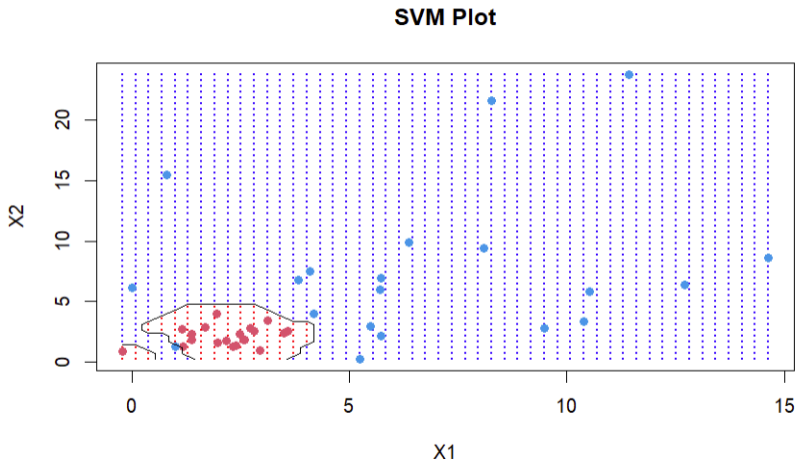
Obviously, this fit is not great. Let's use the same degree and lower the cost to see how that affects our fit.

```
svm.fit <- svm(y ~ ., data = data, degree = 5, kernel =
↪  "polynomial", cost = 1, scale = FALSE)
svm_plot.k(X, y, svm.fit)
```



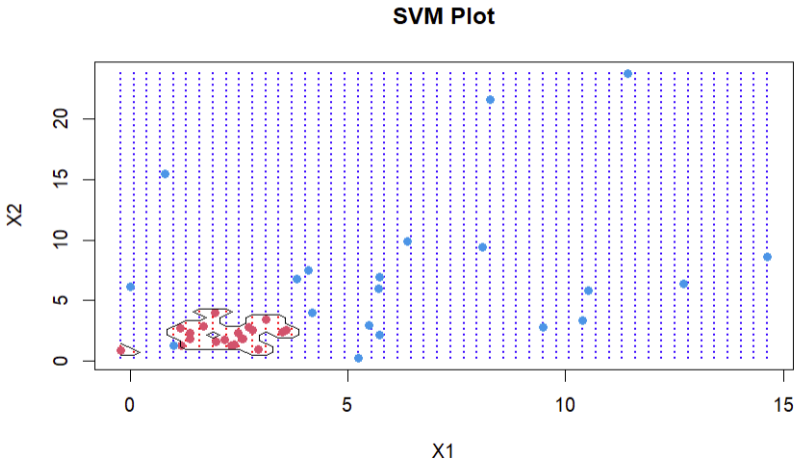
Using a lower cost doesn't provide us as good of a result as we saw with the degree = 3, but it is much better than we had before. Here, we illustrate the challenge of using an SVM with a polynomial kernel: we have 2 choices that influence the bias-variance trade-off, both the C and the degree. We have to try different values for both to see which works best. Now, let's move on to the radial kernel.

```
svm.fit <- svm(y ~ ., data = data, kernel = "kernel", cost = 10,
↪  gamma = 1, scale = FALSE)
svm_plot.k(X, y, svm.fit)
```



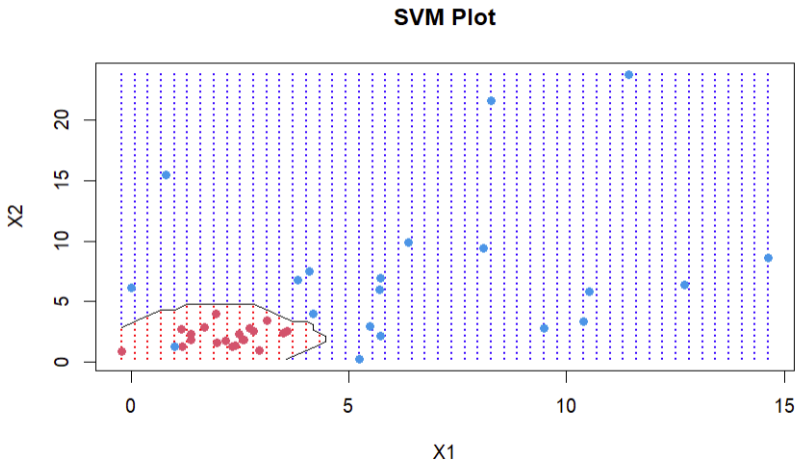
Our first use of the radial kernel with $C = 10$ and $\gamma = 1$ produces an okay result, relatively similar to that of our first polynomial kernel model but with more complexity. Let's try increasing our gamma to see how it impacts our model complexity.

```
svm.fit <- svm(y ~ ., data = data, kernel = "kernel", cost = 10,
↪  gamma = 10, scale = FALSE)
svm_plot.k(X, y, svm.fit)
```



An increase to a γ value of 10 leads to a very complex boundary. Let's see if we can find a better fit by changing our parameters.

```r
svm.fit <- svm(y ~ ., data = data, kernel = "kernel", cost = 5,
↪  gamma = 0.5, scale = FALSE)
svm_plot.k(X, y, svm.fit)
```



We cut the C and the γ in half from our original radial model, and obtain a boundary that captures the classes nicely and is not too complex. We're pretty happy with this result. Overall, the radial kernel can be nicer to use than the polynomial or linear kernel when our boundary can less easily be captured by a simple function of the predictors.

1.4 Conclusion

In conclusion, the exploration of Support Vector Machines (SVMs) reveals their robust and versatile nature, adept at handling both linear and non-linear data separations. From understanding the foundational concepts of maximal margin classifiers and support vector classifiers to the practical applications of SVMs with different kernels, we highlight the algorithm's adaptability. The simulation examples using R solidify the theoretical concepts and also demonstrate the intricate balance between bias and variance, influenced by the choice of kernel and tuning parameters. This exploration underscores the importance of SVMs in the realm of classification tasks and their ability to model complex boundaries efficiently. The adaptability of SVMs in dealing with multiple classes further amplifies their applicability across a wide range of real-world problems.

2 Experimenting with CIFAR-10 and Support Vector Machines

For image classification using Support Vector Machines (SVM), the initial step involves extracting features from the image. These features can encompass various characteristics such as color values of pixels, edge detection results, or even the textures present in the image. Once the features are extracted, they serve as input for the SVM algorithm.

The SVM algorithm operates by identifying the hyperplane that effectively separates distinct classes within the feature space. The fundamental principle of SVMs is to determine the hyperplane that maximizes the margin, the distance between the closest points of different classes. These closest points are referred to as support vectors.

To better understand how image classification works, we will use a real-life dataset and experiment with different parameters using the following steps.

1. **Loading CIFAR-10 Data:** Utilizing Keras, we'll seamlessly load the CIFAR-10 dataset, unraveling its intricacies and understanding the composition of the image classes.

2. **Experimentation with Different Kernels:** SVMs offer flexibility through the choice of kernels. We will experiment with various kernels, including linear, polynomial, and radial basis function (RBF), to observe their impact on classification performance.

3. **Tuning Hyperparameters (C):** SVMs are sensitive to hyperparameters, with 'C' being a crucial factor influencing the trade-off between smooth decision boundaries and accurate classification. We'll experiment with different values of 'C' to understand its effect on model performance.

4. **Evaluating the Model:** Armed with insights into SVMs and the CIFAR-10 dataset, we'll analyze the performance of our chosen model and the optimal parameters. We'll use our analysis of how SVM works and visualize the results using the real-world data.

5. **Comparative Analysis of SVM Strategies:** This section dives into the intricacies of One-vs-One (1-vs-1) and One-vs-All (1-vs-All) strategies, providing insights into their efficacy and trade-offs. Uncover the nuances of accuracy, computational efficiency, and decision boundaries through real-world experimentation on the CIFAR-10 dataset.

6. **Final Results and Conclusions:** This section encapsulates the conclusive evaluation of our SVM model trained on the complete CIFAR-10 training dataset. By employing a linear kernel and regularization parameter C=0.1, we analyze the model's effectiveness across all classes. The focal point is a concise confusion matrix, providing a snapshot of the model's classification accuracy for each category. Through a brief analysis,

we extract insights into the model's strengths and potential areas for improvement, offering a well-rounded conclusion on its overall performance on the training data.

2.1 Loading CIFAR-10 Data

When a computer processes an image, it interprets it as a two-dimensional array of pixels. The dimensions of this array correspond to the image resolution; for instance, if the image is 200 pixels wide and 200 pixels tall, the array will have dimensions of 200 x 200 x 3. The first two dimensions denote the width and height of the image, respectively, while the third dimension represents the RGB color channels. The values within the array range from 0 to 255, signifying the intensity of each pixel at its respective position.

First we would need to import the necessary libraries.

```
import os
import time
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline
```

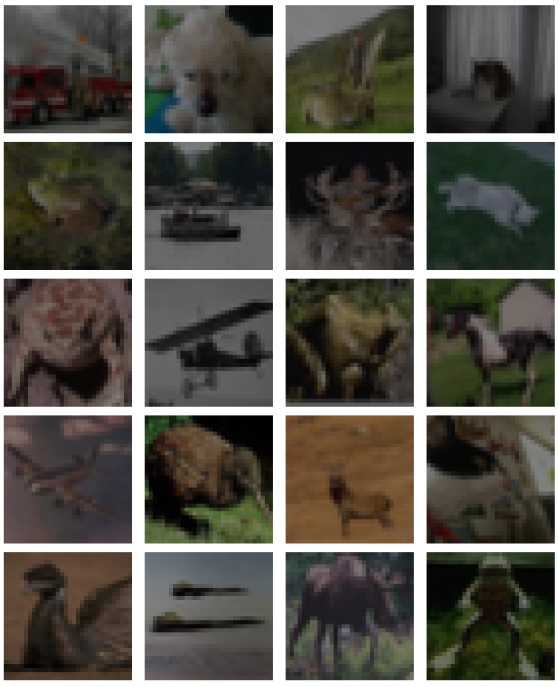
The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 different classes. The data is split into 50,000 training images and 10,000 testing images. We load the data and organize it into training, validation, and test sets.

```
# Load the CIFAR10 dataset
from keras.datasets import cifar10
baseDir = os.path.dirname(os.path.abspath('__file__')) + '/'
classesName = ['plane', 'car', 'bird', 'cat', 'deer', 'dog',
↪  'frog', 'horse', 'ship', 'truck']
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_val = x_train[49000:, :].astype(np.float64)
y_val = np.squeeze(y_train[49000:, :])
x_train = x_train[:49000, :].astype(np.float64)
y_train = np.squeeze(y_train[:49000, :])
y_test = np.squeeze(y_test)
x_test = x_test.astype(np.float64)

# Show dimension for each variable
print ('Train image shape:    {0}'.format(x_train.shape))
print ('Train label shape:    {0}'.format(y_train.shape))
print ('Validate image shape: {0}'.format(x_val.shape))
print ('Validate label shape: {0}'.format(y_val.shape))
print ('Test image shape:     {0}'.format(x_test.shape))
print ('Test label shape:     {0}'.format(y_test.shape))
```

Train image shape: (49000, 32, 32, 3)

Train label shape: (49000,)
Validate image shape: (1000, 32, 32, 3)
Validate label shape: (1000,)
Test image shape: (10000, 32, 32, 3)
Test label shape: (10000,)

```
train_sample = np.random.choice(x_train.shape[0], 20,
↪  replace=False)

fig, axs = plt.subplots(5, 4, figsize=(10, 12))

for i, ax in enumerate(axs.flatten()):
    fig_img = (x_train[train_sample[i], :, :, :] + 1) / 2  #
    ↪  Rescale pixel values to [0, 1]
    ax.imshow(fig_img.astype('uint8'))
    ax.axis('off')

plt.tight_layout()
plt.show()
```



The code above outputs a random set of images from the dataset for us to visualize what the dataset is comprised of and give us a general understanding of the variety of images amongst the 10 classes. Next, we want to normalize the pixel values of our images to a [-1,1] scale and reshape our arrays into vectors.

```python
# Reshaping Data into a Vector and Normalizing it (-1 to 1)
print(x_train.shape)
print(y_train.shape)
x_train = np.reshape(x_train, (x_train.shape[0], -1))
x_val = np.reshape(x_val, (x_val.shape[0], -1))
x_test = np.reshape(x_test, (x_test.shape[0], -1))
print(x_train.shape)
print(x_train[0])

# Normalize
x_train = ((x_train / 255) * 2) - 1
print(x_train.shape)
print(x_train[0])
```

2.2 Experimentation with Different Kernels

This section delves into the exploration of various kernel functions in the context of Support Vector Machines (SVMs) for image classification using the CIFAR-10 dataset. Kernels play a crucial role in SVMs by enabling the algorithm to operate in high-dimensional feature spaces and capture intricate patterns in the data. The objective is to compare the performances of SVMs with different kernels, namely Linear, Polynomial, and Radial Basis Function (RBF), and analyze how each kernel affects the model's ability to discern complex relationships within the image data. This code depicts how to build svm models with the three different types of kernels - linear, radial and polynomial.

```python
# Flatten the images for SVM input
x_train_flat = x_train.reshape(x_train.shape[0], -1)
x_val_flat = x_val.reshape(x_val.shape[0], -1)

# Define SVM classifiers with different kernels
linear_svm = svm.SVC(kernel='linear')
poly_svm = svm.SVC(kernel='poly', degree=3)
rbf_svm = svm.SVC(kernel='rbf')

# Train SVM models
linear_svm.fit(x_train_flat, y_train)
poly_svm.fit(x_train_flat, y_train)
rbf_svm.fit(x_train_flat, y_train)

# Predictions on validation set
linear_preds = linear_svm.predict(x_val_flat)
poly_preds = poly_svm.predict(x_val_flat)
rbf_preds = rbf_svm.predict(x_val_flat)

# Evaluate accuracy on the validation set
linear_acc = accuracy_score(y_val, linear_preds)
```

```
poly_acc = accuracy_score(y_val, poly_preds)
rbf_acc = accuracy_score(y_val, rbf_preds)

# Display results
print('Validation Accuracy - Linear Kernel:
↪  {:.2\%}'.format(linear_acc))
print('Validation Accuracy - Polynomial Kernel:
↪  {:.2\%}'.format(poly_acc))
print('Validation Accuracy - RBF Kernel: {:.2\%}'.format(rbf_acc))
```

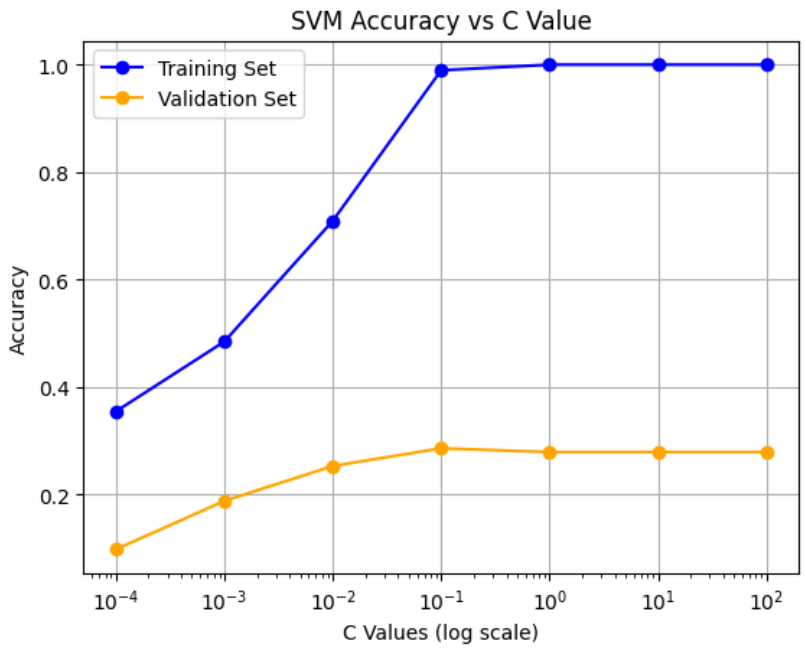
Validation Accuracy - Linear Kernel: 27.90%
Validation Accuracy - Polynomial Kernel: 26.50%
Validation Accuracy - RBF Kernel: 11.90%

2.3 Tuning Hyperparameters (C)

From the previous section we see that the linear kernel has the best accuracy. Linear kernels are effective when the relationship between features and classes is relatively straightforward. By varying the regularization parameter C, we aim to find the optimal balance between fitting the training data well and generalizing to unseen data. This experiment focuses on exploring the impact of different hyperparameter values and kernel functions in Support Vector Machines (SVMs) for image classification, using the CIFAR-10 dataset. SVMs are a classical machine learning technique known for their effectiveness in various tasks, including image classification. The objective is to investigate how the choice of regularization parameter (C) and kernel function influences the model's accuracy. For high values of C, the optimization process tends to favor a hyperplane with a smaller margin, especially if it leads to more accurate classification of all training points. In contrast, when C is set to a very small value, the optimizer seeks a hyperplane with a larger margin, prioritizing the separation distance even if it results in misclassifying more points. We will use a list of c values with a large range (10^{-4} to 10^{2}) to see how much regularisation our model needs.

```
# Initialize lists to store accuracies
acc_train = []
acc_val = []
c_values = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]

# Iterate over different values of C
for c in c_values:
    # Create SVM model with linear kernel
    svc = svm.SVC(probability=False, kernel='linear', C=c)

    # Train the model
    svc.fit(x_train_flat, y_train)

    # Predictions on training set
```

```
    y_pred_train = svc.predict(x_train_flat)
    acc_train.append(accuracy_score(y_train, y_pred_train))

    # Predictions on validation set
    y_pred_val = svc.predict(x_val_flat)
    acc_val.append(accuracy_score(y_val, y_pred_val))

# Plotting the Accuracy for Different C Values on Training and Validation Sets
plt.plot(c_values, acc_train, marker='o', label='Training Set', color='blue')
plt.plot(c_values, acc_val, marker='o', label='Validation Set', color='orange')

plt.xscale('log')  # Using a logarithmic scale for better visibility of smaller C values
plt.xlabel('C Values (log scale)')
plt.ylabel('Accuracy')
plt.title('SVM Accuracy vs C Value')
plt.legend()
plt.grid(True)
plt.show()
```

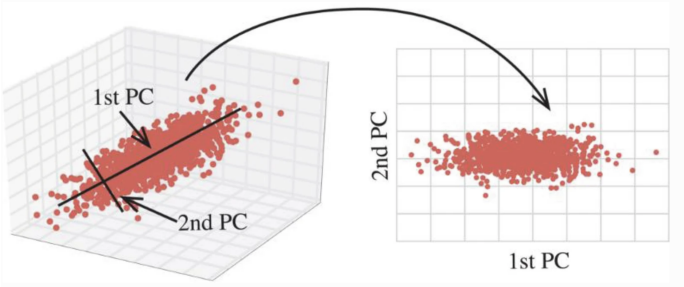


The best results were obtained with a linear kernel SVM for C = 0.1.

2.4 Evaluating the Model using Dimensionality Reduction

The code presented so far leverages Support Vector Machines (SVMs) for binary classification using the CIFAR-10 dataset, a common approach in machine learning for image recognition tasks. A notable aspect of this demonstration involves the visualization of the SVM decision boundary and margins. This visual representation is crucial for interpreting the model's behavior, understanding its generalization capabilities, and making informed decisions during model development.

To visualize the support vector machine classifier we need to use dimensionality reduction using Principal Component Analysis (PCA) to transform the high-dimensional feature space into a lower-dimensional space. Each image has 3072(32*32*3) features, corresponding to the pixel values of the color channels. Reducing the dimensionality to 2 in this case enables visualization of the decision boundary and margins in a 2D plane, facilitating a more intuitive understanding of the SVM's behavior.



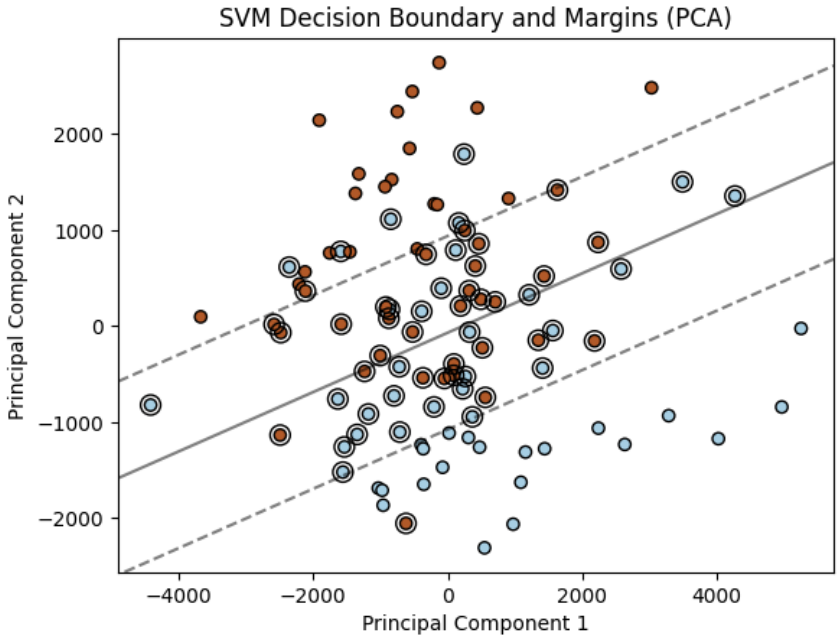
[1]

We also choose two classes because choosing two classes simplifies the problem to binary classification. This focused approach is often used for illustrative purposes and to highlight key concepts, especially when introducing a new technique or algorithm. It allows for a clear demonstration of how Support Vector Machines (SVMs) distinguish between two distinct classes within the CIFAR-10 dataset, providing insights into the model's decision-making process.

```
from sklearn.decomposition import PCA

# Choose two classes for binary classification (e.g., cars and
↪  dogs)
class_indices = [1, 5]  # Class indices for cars and dogs
selected_images = []
selected_labels = []

# Select a subset of images for demonstration
for class_index in class_indices:
    indices = np.where(y_train == class_index)[0][:50]
```

[1] Gupta, Pramod, and Naresh K. Sehgal. "Practical Aspects in Machine Learning." Introduction to Machine Learning in the Cloud with Python

```
    selected_images.append(x_train[indices].reshape(-1, 32 * 32 * 3))
    selected_labels.extend([class_index] * len(indices))

X = np.vstack(selected_images)
y = np.array(selected_labels)

# Apply PCA for dimensionality reduction
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Create an SVM model with a linear kernel and C=0.1
svm_model = svm.SVC(kernel='linear', C=0.1)
svm_model.fit(X_pca, y)

# Plot the decision boundary and margins
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap=plt.cm.Paired, marker='o', edgecolors='k')

# Plot the decision function
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# Create grid to evaluate model
xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 50), np.linspace(ylim[0], ylim[1], 50))
Z = svm_model.decision_function(np.c_[xx.ravel(), yy.ravel()])

# Plot decision boundary and margins
Z = Z.reshape(xx.shape)
plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])

# Highlight support vectors
plt.scatter(svm_model.support_vectors_[:, 0], svm_model.support_vectors_[:, 1], s=100, facecolors='none', edgecolors='k')

plt.title('SVM Decision Boundary and Margins (PCA)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()
```



The plot visualizes the decision boundary, support vectors, and margins of a Support Vector Machine (SVM) trained for binary classification using the CIFAR-10 dataset. The dataset is narrowed down to two classes, allowing for a clear depiction of how the SVM distinguishes between these specific categories. Dimensionality reduction is applied using Principal Component Analysis (PCA), transforming the high-dimensional image feature space into a two-dimensional plane.

In the plot, each point represents an image in the reduced feature space, with different colors indicating the assigned class labels. The decision boundary, depicted as a solid line, separates the two classes, showcasing the SVM's ability to find an optimal hyperplane that maximizes the margin between the classes.

Support vectors, essential data points influencing the position of the decision boundary, are highlighted as large, unfilled circles. These points play a crucial role in determining the robustness and generalization capabilities of the SVM.

Contour lines surrounding the decision boundary represent the margins of the SVM, providing a visual representation of the space where the model makes decisions. Dashed and solid lines represent different margin levels, offering insights into the trade-off between model complexity and generalization.

The overall visualization serves as a comprehensive overview of the SVM's decision-making process, making complex concepts more accessible. It enables the viewer to analyze the model's performance, understand its ability to generalize to unseen data, and gain insights into the impact of support vectors on the decision boundary.

2.5 Multi-class Classification with SVMs

Support Vector Machines (SVMs) are widely utilized for multi-class classification tasks due to their effectiveness in high-dimensional feature spaces. Two common strategies for extending binary SVMs to multi-class problems are One-vs-One (1-vs-1) and One-vs-All (1-vs-All). In this experiment, we aim to compare the classification performance and computational efficiency of these strategies using the CIFAR-10 dataset, a well-known collection of images categorized into ten distinct classes.

Support Vector Machines (SVMs) are robust classifiers commonly employed for multi-class classification tasks, offering versatility and powerful discriminative capabilities. Two prevalent strategies, One-vs-One (1-vs-1) and One-vs-All (1-vs-All), extend SVMs to handle multiple classes. In the 1-vs-1 approach, binary classifiers are trained for every possible pair of classes, creating a multitude of decision boundaries. This method is advantageous for potentially avoiding imbalances in class distributions but may incur increased training time with a large number of classes. On the other hand, the 1-vs-All strategy simplifies the problem by training a binary classifier for each class against the rest. While it reduces the number of classifiers to C, where C is the number of classes, it may encounter challenges with imbalanced class distributions and indirect decision boundaries. This experiment aims to delve into the efficiency of these strategies, evaluating their classification accuracy and computational performance on the CIFAR-10 dataset, ultimately providing insights for selecting an optimal approach based on the specific characteristics of the data and computational constraints.

```
# Standardize features
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_val = scaler.transform(x_val)
x_test = scaler.transform(x_test)

# Train and evaluate 1-vs-1 SVM
start_time_svm_ovo = time.time()
svm_ovo = OneVsOneClassifier(SVC(kernel='linear', C=0.1))
svm_ovo.fit(x_train, y_train)
svm_ovo_predictions = svm_ovo.predict(x_val)
accuracy_svm_ovo = accuracy_score(y_val, svm_ovo_predictions)
end_time_svm_ovo = time.time()
print(f'1-vs-1 SVM accuracy on validation set:
↪  {accuracy_svm_ovo:.4f}')
print(f'Time taken by 1-vs-1 SVM: {end_time_svm_ovo -
↪  start_time_svm_ovo:.2f} seconds')

# Train and evaluate 1-vs-All SVM
start_time_svm_ovr = time.time()
svm_ovr = OneVsRestClassifier(SVC(kernel='linear', C=0.1))
```

```
svm_ovr.fit(x_train, y_train)
svm_ovr_predictions = svm_ovr.predict(x_val)
accuracy_svm_ovr = accuracy_score(y_val, svm_ovr_predictions)
end_time_svm_ovr = time.time()
print(f'1-vs-All SVM accuracy on validation set:
↪  {accuracy_svm_ovr:.4f}')
print(f'Time taken by 1-vs-All SVM: {end_time_svm_ovr -
↪  start_time_svm_ovr:.2f} seconds')
```

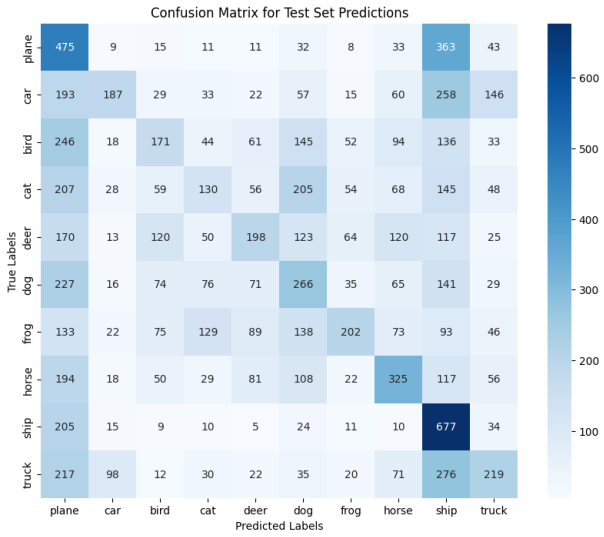
1-vs-1 SVM accuracy on validation set: 0.2630
Time taken by 1-vs-1 SVM: 66.19 seconds
1-vs-All SVM accuracy on validation set: 0.2300
Time taken by 1-vs-All SVM: 164.17 seconds

The results of the experiment indicate that the One-vs-One (1-vs-1) SVM strategy achieved an accuracy of 26.30% on the validation set, while the One-vs-All (1-vs-All) SVM strategy achieved an accuracy of 23.00%. The 1-vs-1 SVM outperforms the 1-vs-All SVM in terms of classification accuracy, suggesting its efficacy in capturing intricate class relationships. However, it's noteworthy that this improvement in accuracy comes with a trade-off in computational efficiency. The training and prediction times for the 1-vs-1 SVM were significantly shorter, with a total of 66.19 seconds, compared to the 1-vs-All SVM, which took 164.17 seconds. This shows that there could be an inherent trade-off between accuracy and computational efficiency in SVM strategies. Practitioners need to carefully consider these trade-offs based on the specific requirements of their applications, particularly when dealing with large datasets or constrained computational resources.

2.6 Final Results and Conclusions

In this section, we wrap up our SVM model exploration by examining its performance on the complete CIFAR-10 training dataset. Using a linear kernel and C=0.1, we present a practical overview of the model's classification accuracy across all categories. Here's a sample of the test images that were predicted by the svm model. Some were predicted correctly, but the overall accuracy of the model was not very high.

The key highlight is a straightforward confusion matrix, providing a quick snapshot of where the model excels and areas for potential improvement.



Our brief analysis aims to distill practical insights, offering a conclusive perspective on the SVM's effectiveness in handling the diverse set of CIFAR-10 images. This method could be applied to any multiclass image classification, but the most optimal parameters will vary from dataset to dataset. Overall, SVM's are a useful classical machine learning method for image classification.

3 An Exploration of Convolutional Neural Networks

3.1 Basis of Neural Networks

Neural networks are another type of model that takes in an input of p variables and uses a nonlinear function $f(X)$ to predict the response y. Neural networks share the same general structure: an input layer, one or more hidden layers, and an output layer. Each layer has a series of nodes which are interconnected between layers. Mathematically, the simple model can be represented using the equation

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$

K represents the number of nodes in the hidden layer. Each of the K nodes has an activation A_k, k = 1,...,K:

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$

g(z) is a nonlinear activation function, which is applied to the weighted sum of inputs at each node in the layer and introduces nonlinearity to the model. In a single hidden layer neural network, these activations would then feed into the output layer. Common activation functions are the sigmoid activation function and the rectified linear unit (ReLU) activation function. The sigmoid activation function was favored in earlier instances of neural networks and holds the form

$$g(z) = \frac{e^z}{1 + e^z}$$

The ReLU activation function is the modern choice for an activation function and is structured as such

$$ReLU(z) = max(0, z)$$

The form of a neural network with more than one hidden layer is very similar but the activations outputted from the first hidden layer will act as the input to the second hidden layer and so forth until the output layer is reached. Commonly, when classifying images, the output layer will return the probability of the inputted image being in each class. So, for any m class,

$$f_m(X) = \beta_{m0} + \sum_{k=1}^{K} \beta_{mk} A_k$$

Then, by applying the softmax activation function, the outputs will be probabilities.

$$f_m(X) = Pr(Y = m|X) = \frac{e^{z_m}}{\sum_{k=0}^{m} e^{z_k}}$$

3.2 Convolutional Neural Networks

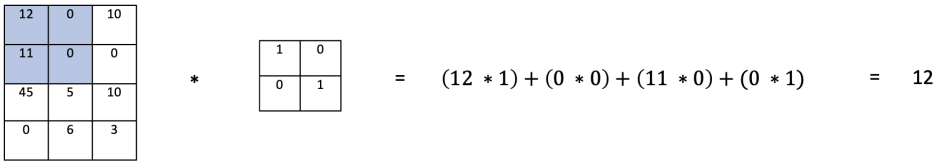
Convolutional Neural Networks (CNNs) are a specific type of neural network that has shown success in classifying images. The basis of CNNs is identifying low-level features in an image (color, small edges, etc.) and then combining these features to form higher-level features. The class of the image is determined using the presence of the higher-level features and the class with the highest probability of being true is chosen.

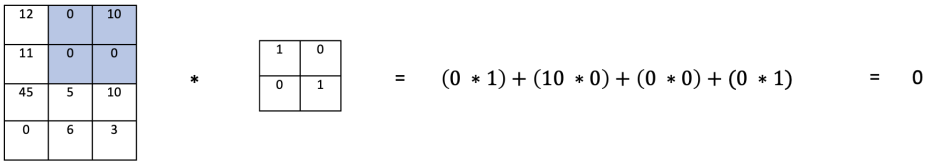
Images can be broken down into a set of pixels, each with their own eight-bit RGB value. Represented in a mathematical sense, an image is a set of three stacked matrices with each matrix either storing the red, green, or blue values of the pixels. This structure is called a feature map; it is a three-dimensional array where the first two axes are the dimensions of the image and the third axis is for the channels, described later. These matrices are the input to the neural network.

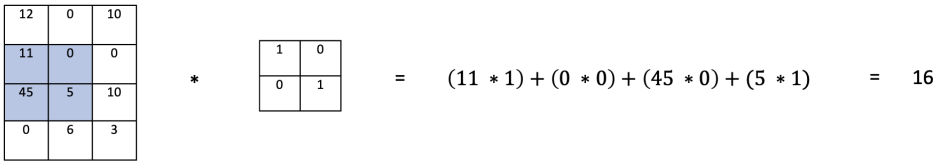
CNNs consist of two types of layers: convolution layers and pooling layers. Convolution layers are made up of many convolution filters. A convolution filter

is a small matrix of numbers that is passed over an image and used to determine whether a particular feature is present in the image. This process occurs through convolution in which the convolution filter is repeatedly multiplied by the elements in a submatrix of the image matrix, and the results are summed together. If the submatrix of the inputted image resembles the convolution filter, the sum of the results will have a large value; otherwise, it will have a small value. The formula for the convolution is included below where the input image is denoted by f and the kernel by h. m is the indexes of the rows and n of the columns of the outputted matrix.

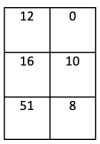
$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k]$$





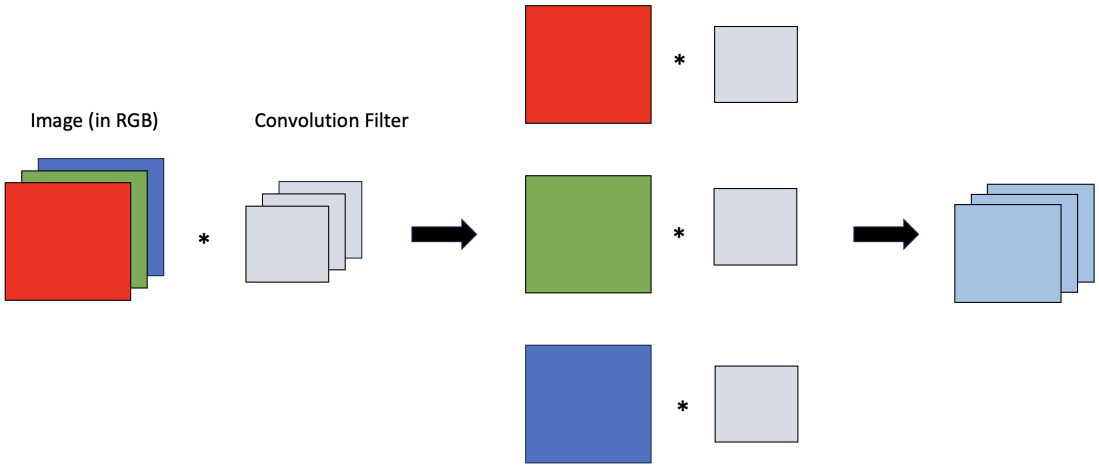


Result of convolution:



The figures above are a very simple example of how the convolution works. As mentioned before, an image typically has color on the RGB scale and thus

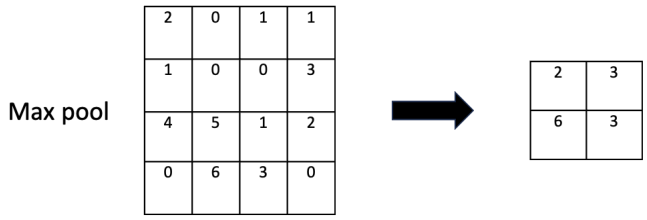
has a matrix or channel for each color. Thus, the convolution filter will also have the same number of channels as the inputted image.



As seen in both of the examples above, each time a convolution is performed, the resulting image is smaller than the image inputted. This shrinking limits the number of times convolution can be performed. Furthermore, convolution is better at capturing information closer to the center of the image than at the edges of the image. Fortunately, the images can be padded with additional pixels around its edges. This allows the inputted image and the outputted image to have the same dimensions. The width of the padding can be determined using the following equation, where p is padding and f is the filter dimension:

$$p = \frac{f-1}{2}$$

The other type of layer is pooling layers. They are used to reduce the size of the image into a smaller summary image and speed up calculations. In the pooling layers, the image is divided into regions and a certain operation is performed on each region. For example, the max pooling operation summarizes each different region of the image by returning the maximum value of that region. Below is an example of max pooling for every 2x2 non-overlapping block of pixels in the matrix.



3.3 Fitting a CNN

To fit a good CNN, the model parameters must be calculated. Since we are looking into the classification of images, to compare the outputted predictions to the actual results we look at cross-entropy

$$L = -\sum y_i log(f(x_i))$$

Where y is the true probability of the image pertaining to the class and $f(x)$ is the CNNs predicted probability distribution. To train the neural network, we want to find parameter values that minimize the cross-entropy. A way of estimating these parameters is through feedforward and backpropagation.

3.3.1 Feedforward

Combining our convolution layers and pooling layers, we can start building the neural network. When the image I gets inputted, through convolution it will be multiplied by a convolution filter k. Bias b will then be added to the resulting output, and the new matrix will be passed through the activation function (ReLU).

$$S = \sum \sum Ik + b$$

$$C = ReLU(S)$$

It will continue through the neural network to the pooling layer, and the image will get reduced by the max pooling function.

$$P = max(C)$$

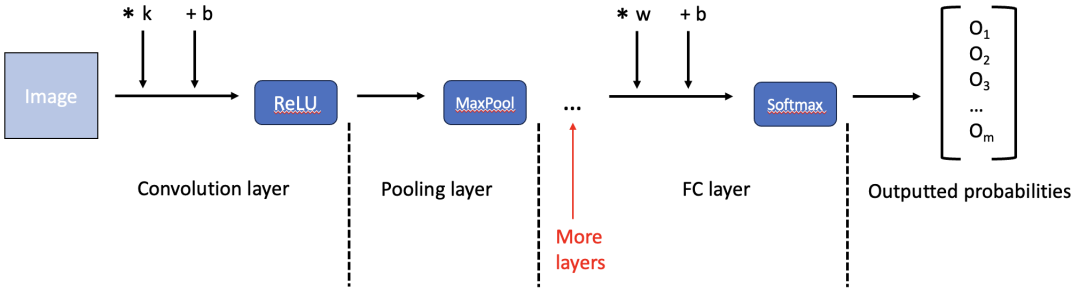
The steps through the convolution and pooling layers will repeat for a defined amount and then reach the fully-connected (FC) layer. In the fully connected layer, all the nodes will be connected to all the activations in the previous layer. The FC layer calculates the probability of the image identifying in each class using the softmax activation function, as described earlier.

$$f = flatten(P^2)$$

$$S = \sum_w f + B$$

$$O = softmax(S)$$

Where w are weights and B is bias. Below is an example of a compiled CNN.



3.3.2 Backpropogation

To find the optimal values of the parameters, we minimize the cross-entropy equation by working backwards through the neural network using the Stochastic Gradient Descent algorithm. Gradient descent is an iterative optimization algorithm that allows you to find the minimum of the cross-entropy function by continuously calculating the partial derivatives or gradient and updating the parameters. Stochastic Gradient Descent is very similar but instead of using the whole dataset to compute the gradient and the parameters, it uses random small subsets of the data or minibatches. To calculate the partial derivatives, the chain rule is used.

Starting with the FC layer, the first step is to derive the gradients for bias and the weights:

$$\frac{dL}{dB}, \frac{dL}{dw}$$

Then, working backwards, for the next convolution and pooling layer, we calculate the gradients for the bias and the kernel

$$\frac{dL}{db}, \frac{dL}{dk}$$

We repeat the derivations for the other convolution and pooling layers until we have completely derived the gradients for all the layers. Once, we have all the gradients, we can update the parameters as such:

$$k^1 = k^1 - \eta \frac{dL}{dk^1}$$

$$b^1 = b^1 - \eta \frac{dL}{db^1}$$

$$\dots$$

$$k^d = k^d - \eta \frac{dL}{dk^d}$$

$$b^d = b^d - \eta \frac{dL}{db^d}$$

$$w = w - \eta \frac{dL}{dw}$$

$$B = B - \eta \frac{dL}{dB}$$

Where η is the learning rate and d is the number of convolution/pooling layers.

3.4 Simulated Data Example

To study the CNN's performance, we can craft an experiment by inputting basic shapes into a simple CNN. We do this by first defining functions which create pixel images of circle, squares, and triangles, the first function, and then add noise to the images based on a adjustable noise level, second function. The noise is added by flipping random pixels in the image, from 0 to 1 and vice versa. The noise level determines the number of pixels that get flipped.

```
generate_image <- function(shape, height = 28, width = 28) {
  image <- matrix(0, nrow = height, ncol = width)

  switch(shape,
         circle = {
           center <- c(height / 2, width / 2)
           radius <- sample(3:7, 1)
           grid <- expand.grid(x = 1:height, y = 1:width)
           mask <- sqrt((grid$x - center[1])^2 + (grid$y -
           ↪  center[2])^2) < radius
           image[cbind(grid$x, grid$y)[mask,]] <- 1
         },
         square = {
           side <- sample(8:12, 1)
           start <- sample(1:(height - side), 2)
           image[start[1]:(start[1]+side),
           ↪  start[2]:(start[2]+side)] <- 1
         },
```

```
        triangle = {
          size <- sample(8:12, 1)
          start <- sample(1:(height - size), 1)
          for (i in 0:size) {
            image[start + i, (width/2 - i/2):(width/2 + i/2)] <- 1
          }
        }
  )
  image
}

noise <- function(shape, height = 28, width = 28, noise_level =
↪  0.50) {
  image <- generate_image(shape, height, width)
  total_pixels <- height * width
  num_noise_pixels <- round(total_pixels * noise_level)
  noise_pixels <- sample(total_pixels, num_noise_pixels)
  for (pixel in noise_pixels) {
    row <- ((pixel - 1) %/% width) + 1
    col <- ((pixel - 1) %% width) + 1
    image[row, col] <- 1 - image[row, col]
  }

  image
}
```

Using the keras and tensorflow packages in R, we can build a CNN model with one convolution layer, one pooling layer, and one fully connected layer. The convolution layer uses the ReLU activation function and a 3x3 kernel. The pooling layer uses the max pooling operation to then reduce the image. In the final layer, the CNN uses the softmax function to output the probability of the image being classified as each shape. Since it is a classification task, the proper loss function is cross-entropy. The CNN returns the accuracy of the outputted prediction compared to the true labels.

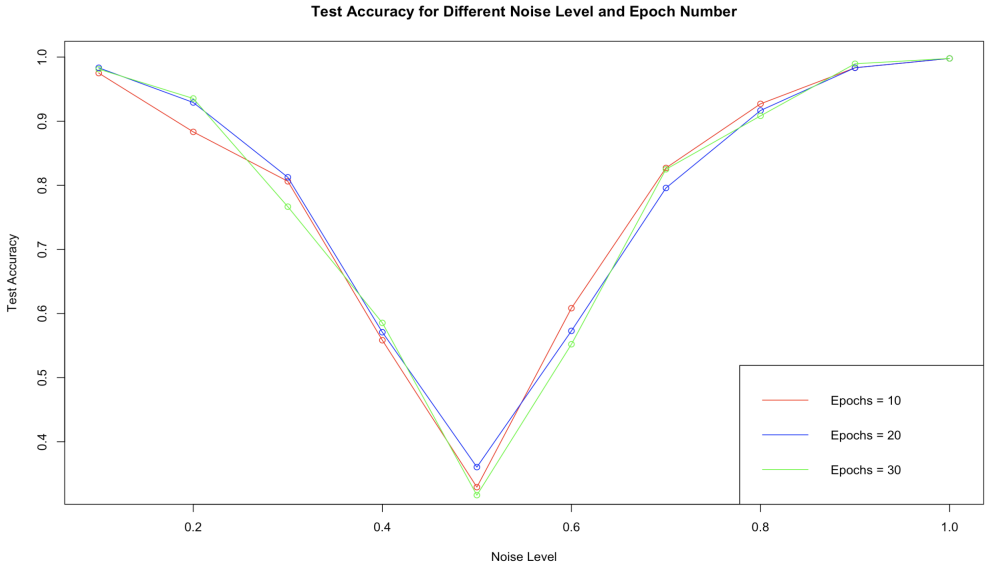
```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation =
  ↪  'relu',
                input_shape = c(28, 28, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = length(shapes), activation = 'softmax')

model %>% compile(
  loss = 'sparse_categorical_crossentropy',
  optimizer = 'adam',
  metrics = c('accuracy'))
```

To test the performance of this CNN model, you can run a for loop adjusting the noise level to 0.1, 0.2, ..., 1 and setting the number of epochs to 10, 20, and 30. The number of epochs is the number of times the training data is passed through the model, both forwards and backwards, to train the model. The goal is to see which values for these parameters lead to best test accuracy of the model.

```
noises <- seq(0.1,1, by = 0.1)

epochs_list <- c(10,20,30)

test_accuracy <- matrix(data=NA,nrow=length(noises),ncol=length(epochs_list))

m = 1

for (ns in noises){
  j = 1
  print(i)

  for (epcs in epochs_list){
      set.seed(1)
    n <- 3000
    shapes <- c("circle", "square", "triangle")

    images <- array(0, dim = c(n, 28, 28, 1))
    labels <- integer(n)

    for (i in 1:n) {
      shape <- sample(shapes, 1)
      images[i, , , 1] <- noise(shape, noise_level = ns)
      labels[i] <- match(shape, shapes) - 1
    }

    train_indices <- sample(1:n, size = 0.8 * n)
    train_images <- images[train_indices, , , ]
    train_labels <- labels[train_indices]

    test_images <- images[-train_indices, , , ]
    test_labels <- labels[-train_indices]

    model <- keras_model_sequential() %>%
      layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu',
                    input_shape = c(28, 28, 1)) %>%
      layer_max_pooling_2d(pool_size = c(2,2)) %>%
      layer_flatten() %>%
      layer_dense(units = 128, activation = 'relu') %>%
```

```
      layer_dense(units = length(shapes), activation = 'softmax')

    model %>% compile(
      loss = 'sparse_categorical_crossentropy',
      optimizer = 'adam',
      metrics = c('accuracy')
    )

    CNN_results <- model %>% fit(
      train_images, train_labels,
      epochs = epcs,
      validation_split = 0.2
    )

    test_accuracy[m,j] <- tail(CNN_results$metrics$val_accuracy, 1)

    j = j + 1

  }

  m = m + 1
}

```

```
plot(noises,test_accuracy[,1], xlab = 'Noise Level', ylab = 'Test Accuracy', main = 'Test Accuracy for Different Noise Level and Epoch Number', col = 'red')
lines(noises,test_accuracy[,1], col = 'red')
points(noises, test_accuracy[,2], col = 'blue')
lines(noises,test_accuracy[,2], col = 'blue')
points(noises,test_accuracy[,3], col = 'green')
lines(noises,test_accuracy[,3], col = 'green')
legend("bottomright", legend = c("Epochs = 10", "Epochs = 20", "Epochs = 30"), col = c("red", "blue", "green"), lty = 1)
```



Looking at the plot above, it can be seen that while the number of epochs does not cause big differences in the test accuracy, the amount of noise in the data does cause drastic differences in the test accuracy. The test accuracy is very high, almost 100 percent correct classification, when the noise levels are close to 0 or close to 1. This can be explained by the fact that when a very small proportion of pixels in the image are flipped, the image remains close to the general shape and the CNN does well at classifying the image. When a large proportion of pixels are flipped, the general structure of the shape remains the same but the colors/pixel values are inverted, which would also make it easier for the CNN to identify the correct class. However, when the noise level is around 0.5, meaning 50 percent of the pixels get flipped, the CNN struggles to identify the more heavily mutated shape, and the test accuracy is very low. Even when the simulated data set was increased from 3,000 rows to 10,000 rows, the plot looked extremely similar to the one above. This suggests that for even more simple images, a neural network will struggle to correctly classify images that are far from the expected shape without correct training and modification of the neural network. In future exploration, it would be interesting to add more convolution and pooling layers to the neural network and adjust the parameters further to increase the accuracy when using noisy data.

4 Experimenting with Convolutional Neural Networks Using CIFAR-10

We now analyze a real-world example with data from the University of Toronto's **CIFAR-10 dataset**. The dataset contains a total of $60,000$ 32×32 images with initialized train-test splits of $50,000$ training images and $10,000$ test images.

Each image is to be classified into one of 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each class consists of $6,000$ images with $5,000$ and $1,000$ in the training and test data splits, respectively. We aim to construct a SVM for $k = 10$ classes and a CNN with optimally tuned hyperparameters, adjusting the batch size, number of epochs, learning rate, loss metric, and optimizer. Then, we seek to compare the performances of the models through metrics including accuracy and cross-entropy loss.

4.1 Imports

For this section, we will use Python to load the data and train our models. We require the PyTorch machine learning framework library, specifically the following imports:

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import time
import math
```

We use `torchvision` to transform the raw data into PyTorch tensors, multi-dimensional objects representing the input data fed into neural networks. We use `torch.nn` for building the layers of the model. We use `matplotlib` for plotting our results and `time` for tracking training time to analyze models of different batch sizes.

4.2 Loading the Dataset

We first need to load the CIFAR-10 dataset into our Python environment. To obtain the dataset, navigate to https://www.cs.toronto.edu/~kriz/cifar.html and download the CIFAR-10 Python version. This will give us access to a directory that we can place in our `src` directory of our project. [add image here]

We can now write a function to retrieve this data and convert it into PyTorch tensors for our use. Before we begin with loading the dataset, we should select a normalization scheme for our data. The reason we normalize the data is to keep different features on a more similar scale.

We run into a fork in the road, where we may choose from a variety of schemes for which to normalize the data. Some common choices include

- Standard normalization: $\mu = (0, 0, 0)$, $\sigma = (1, 1, 1)$
- ImageNet normalization: $\mu = (0.485, 0.456, 0.406)$, $\sigma = (0.229, 0.224, 0.225)$

- Inception normalization: $\mu = (0.5, 0.5, 0.5)$, $\sigma = (0.5, 0.5, 0.5)$

We observe 3 values since each pixel of the image represents an length-3 vector encoding of RGB values from 0 to 255. The normalized values will be much closer to one another and increase stability in the model. For this tutorial, we will stick to **standard normalization**.

Now, we can begin initializing the datasets. We need to split the samples accordingly into smaller batches for our model to process one at a time. Specifically, we must provide a **batch size**, the parameter that controls the number of samples that are processed by the model before updating parameters, which can impact the training time and performance. Smaller batch sizes will likely take longer to run due to a large amount of parameter updates but can result in a better model than larger batch sizes.

To get started, we read the `torchvision.datasets.CIFAR10`, then initialize `DataLoader` objects that load the data. We set the `shuffle` argument to True so that the data samples are randomly ordered after every epoch.

```
def load_cifar10(batch_size):
    # Convert into tensors and normalize by mean=0.0, stdev=1.0
    transform = transforms.Compose([transforms.ToTensor(),
    ↪  transforms.Normalize((0.0, 0.0, 0.0), (1.0, 1.0, 1.0))])

    # Load training data
    train_set = torchvision.datasets.CIFAR10(root='./data',
    ↪  train=True, download=True, transform=transform)
    train_loader = torch.utils.data.DataLoader(train_set,
    ↪  batch_size=batch_size, shuffle=True, num_workers=2)

    # Load test data
    test_set = torchvision.datasets.CIFAR10(root='./data',
    ↪  train=False, download=True, transform=transform)
    test_loader = torch.utils.data.DataLoader(test_set,
    ↪  batch_size=batch_size, shuffle=False, num_workers=2)

    # Classes representing the labels 0-9
    classes = ('airplane', 'automobile', 'bird', 'cat', 'deer',
    ↪  'dog', 'frog', 'horse', 'ship', 'truck')

    return train_loader, test_loader, classes
```

Now that the data has been loaded, we are ready to proceed with defining the model architecture.

4.3 Model

To define our basic model layout, we must create convolutional, max pooling, and fully-connected layers. The convolutional layers stride across the tensor and apply element-wise multiplication between a selected kernel and the current

position within the convoluted image. The max pooling layers are important for dimensionality reduction and decreasing noise by taking the max value stored in non-overlapping square matrices for each of the RGB layers. The fully-connected layers operate following the flattening of the tensor to a linear structure by augmenting the number of relevant features through a linear transformation $y = xW^T + b$, where W is the matrix of weights and b is the bias.

We begin by creating a CNN class to modularize our code and initialize the layers in its constructor:

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)  # convolutional layer 1
        self.conv2 = nn.Conv2d(6, 12, 5)  # convolutional layer 2
        self.pool = nn.MaxPool2d(2, 2)  # max pooling layer
        self.fc1 = nn.Linear(12 * 5 * 5, 120)  # linear layer 1
        self.fc2 = nn.Linear(120, 64)  # linear layer 2
        self.fc3 = nn.Linear(64, 10)  # linear layer 3

    ...
```

In PyTorch, the `nn.Conv2d` layer is fed in the parameters (num. input channels, num. output channels, kernel dimension). **Channels** represent mediums of which to characterize the data, like for example, for image classification, we typically begin with the 3 mediums of RGB. The number of channels is expanded during the first part of the CNN feed-forward propagation. Furthermore, the layer by default uses a stride length of 1, meaning matrix multiplication occurs for every consecutive window (can be overlapping), and it does not add any additional padding to the tensors, equivalent to **valid padding**. The `nn.MaxPool2d` layer is fed in the parameters (row dimension, column dimension), so in this case, it utilizes a 2×2 matrix to extract the max values and suppress noise. The `nn.Linear` layers are fed in the parameters (num. input features, num. output features). Ultimately, our goal is to output 10 features representing the classes in CIFAR-10, so the linear layers work to reduce the number of features in the second half of the algorithm.

Now, let's put all of these layers to use and construct a fully-functioning model.

```
class CNN(nn.Module):
    ...

    def forward(self, x):
        x = self.pool(func.relu(self.conv1(x)))
        x = self.pool(func.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = func.relu(self.fc1(x))
```

```
9           x = func.relu(self.fc2(x))
10          x = self.fc3(x)
11          return x
```

We now see how the layers' parameters come to fruition. The image is convoluted to a smaller size with the extraction of more channels after every convolutional layer, and the pooling layer further reduce the size by half. Specifically, from $4 \times 3 \times 32 \times 32$ as the input given a batch size of 4, 3 channels representing the RGB values, and 32×32 pixel images, the tensor is reshaped to $4 \times 6 \times 14 \times 14$ after line 5, and $4 \times 12 \times 5 \times 5$ after line 6. This is because with valid padding, generating a value for every element-wise multiplication with a $k \times k$ kernel will reduce the dimensions of the image by $k - 1$, and the pooling layer that follows cuts the dimensions further in half. Once the tensor is flattened, we obtain vectors of length $12 \cdot 5 \cdot 5 = 300$ for each sample, whose features are then reduced to 10 via the linear layers.

For this tutorial, we will stick to this architecture for the layers of the neural network and adjust other hyperparameters. Now, we will begin training the model.

4.4 Training the Model

We now train this model as we would any neural network. In addition to the training process, we keep track of the loss values for every $2,000$ batches fed into the model and the time it takes to train the model.

In order to achieve faster convergence on the optimal solution, we run the model over several epochs of the training data. An **epoch** is an iteration over all samples of the dataset. Within each epoch, we iterate through the batches of our dataset with the following steps:

1. Zero the gradients to ensure that no unnecessary information is held in the parameters for each fresh batch.
2. Run the CNN (forward propagation of the inputs through the layers).
3. Compute the loss at each step.
4. Run backward propagation to the start of the neural network layer structure for the next iteration, and update the gradients and the overall loss.

We write the function `train_model` that handles all of the above specifications for model training:

```
1 def train_model(cnn, train_loader, criterion, optimizer, epochs):
2     loss_vals = []
3     start_time = time.time()
4 
5     # Run the model through the dataset "epochs" times
6     for epoch in range(epochs):
```

```
        running_loss = 0.0
        for i, data in enumerate(train_loader, 0):
            # Unpack inputs and labels from the training data
            inputs, labels = data

            # Zero gradients
            optimizer.zero_grad()

            # Run forward propagation
            outputs = cnn(inputs)

            # Compute the loss function
            loss = criterion(outputs, labels)

            # Run backward propagation
            loss.backward()
            optimizer.step()

            # Print the loss for every 2000 batches
            running_loss += loss.item()
            if i % 2000 == 1999:
                print(f'[{epoch + 1}, {i + 1:5d}] loss:
                ↪  {running_loss / 2000:.3f}')
                loss_vals.append(running_loss / 2000)
                running_loss = 0.0

    print('Finished training.')

    # Get training time of the model
    training_time = time.time() - start_time
    print(f'Training time: {training_time}')

    return loss_vals, training_time
```

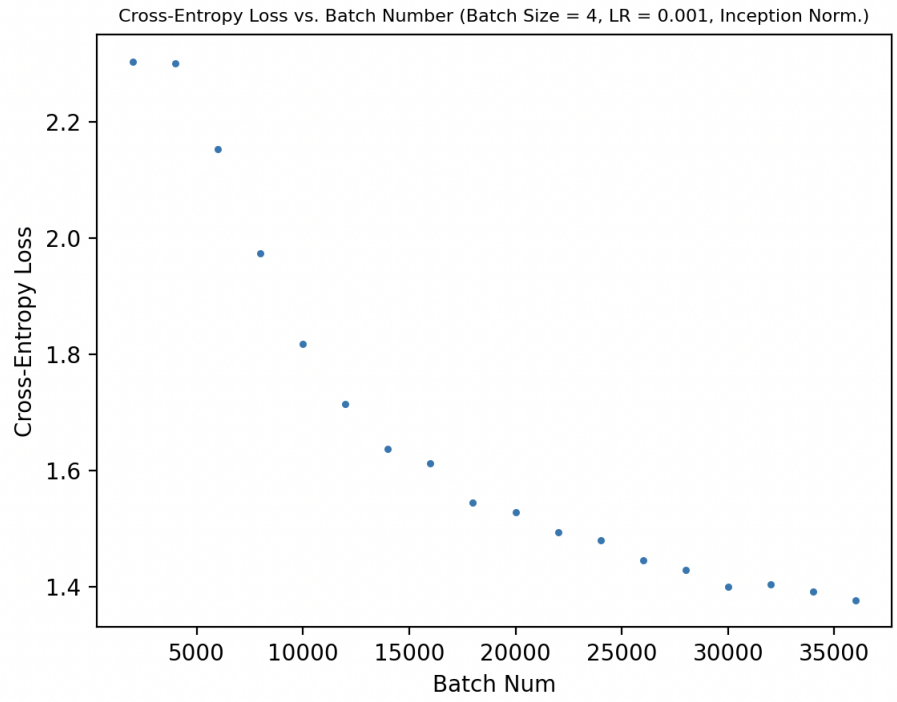
For our initial example, we use a batch size of 4 for loading in the data, cross-entropy loss as our measurement of performance, and 3 epochs during training. Note that we can select a reasonable number of epochs based on experimenting with varying epoch values and determining if model loss begins to flatten or if the model begins to overfit, which we will do later in our analysis. Moreover, we track our training loss visually by plotting it against the number of batches.

```
if __name__ == '__main__':
    # Load the dataset
    train_loader, test_loader, classes = load_cifar10(batch_size=4)

    # Initialize the neural network
    cnn = CNN()

```

```
    # Define the criterion for measuring performance
    criterion = nn.CrossEntropyLoss()

    # Define the optimizer for the model
    optimizer = optim.SGD(cnn.parameters(), lr=1e-3, momentum=0.9)

    # Train the model
    epochs = 3
    loss_vals, _ = train_model(cnn, train_loader, criterion,
    ↪  optimizer, epochs)
    plt.scatter([2000 * i for i in range(1, 6 * epochs + 1)],
    ↪  loss_vals, s=5)
    plt.title("Cross-Entropy Loss vs. Batch Number (Batch Size = 4,
    ↪  LR = 0.001, Inception Norm.)",
              fontsize=8)
    plt.xlabel("Batch Num")
    plt.ylabel("Cross-Entropy Loss")
    plt.show()
```

```
[1,  2000] loss: 2.303
[1,  4000] loss: 2.301
[1,  6000] loss: 2.153
[1,  8000] loss: 1.974
[1, 10000] loss: 1.818
[1, 12000] loss: 1.715
[2,  2000] loss: 1.638
[2,  4000] loss: 1.612
[2,  6000] loss: 1.545
[2,  8000] loss: 1.527
[2, 10000] loss: 1.493
[2, 12000] loss: 1.480
[3,  2000] loss: 1.445
[3,  4000] loss: 1.429
[3,  6000] loss: 1.400
[3,  8000] loss: 1.404
[3, 10000] loss: 1.392
[3, 12000] loss: 1.377
Finished training.
Training time: 154.01734495162964
```

Output from running `train_model`.



Plot of training cross-entropy loss vs. batch number for initial CNN example.

4.5 Testing the Model

Now that the model is trained, it is important to test its ability to make accurate predictions on data it has not seen yet to test for overfitting. We determine the test accuracy by feeding the CNN our test split and comparing the predicted labels with the ground truth labels.

```
def test_model(cnn, test_loader, classes):
    total_correct = {obj_class: 0 for obj_class in classes}

    with torch.no_grad():
        for data in test_loader:
            # Unpack inputs and labels from the test data
            images, labels = data

            # Run the model on test data
            outputs = cnn(images)

            # Get the highest likelihood predicted class
            _, preds = torch.max(outputs, 1)

```

```
            # Update total correct if predictions match actual label
            for actual, pred in zip(labels, preds):
                if actual == pred:
                    total_correct[classes[actual]] += 1

    total_samples = 10000
    accuracy = float(sum(total_correct.values()) / total_samples)
    print(f'Accuracy on test data: {accuracy}')
    print('Breakdown of classes: ')
    for obj_class, correct in total_correct.items():
        print(f'Accuracy of {obj_class}: {float(correct / (0.1 * total_samples))}')

    return accuracy
```

```
Accuracy on test data: 0.5214
Breakdown of classes:
Accuracy of airplane: 0.502
Accuracy of automobile: 0.753
Accuracy of bird: 0.462
Accuracy of cat: 0.541
Accuracy of deer: 0.426
Accuracy of dog: 0.159
Accuracy of frog: 0.659
Accuracy of horse: 0.502
Accuracy of ship: 0.673
Accuracy of truck: 0.537
```

Output from running `test_model`.

We can see that our model predicted the correct class 52% of the time, which is much better than random chance of 10% but not necessarily the best performance. Now, we can turn to tuning the model and determining if we find hyperparameters that increase performance.

4.6 Hyperparameter Tuning

We will continue exploring our model through the following means: batch size, number of epochs, learning rate, loss metric, and optimizer. For this section, when we analyze one of the above, we will hold the others constant, using the default values of batch size = 4, number of epochs = 3, learning rate = $5 \cdot 10^{-4}$, and `optim.SGD` as the optimizer.

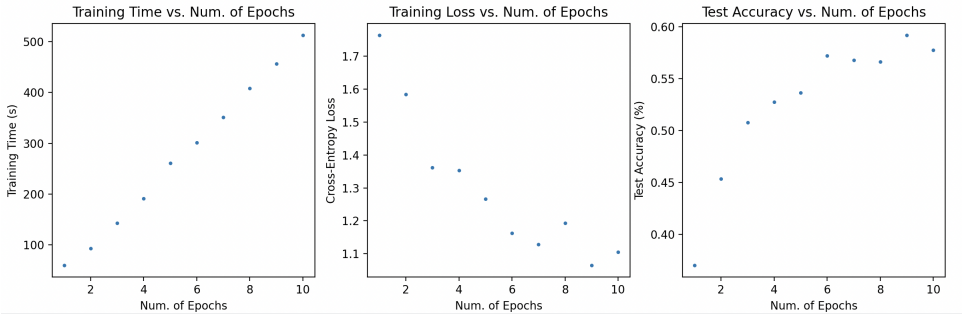
4.6.1 Batch Size: Training Time vs. Performance

We can plot training time vs. number of epochs, as well as training loss and test accuracy.

We observe that with larger batch sizes, the training time is much lower. This is sensable since the more samples within a batch that are processed simultaneously by the model, the less iterations the model goes through. However, we also notice that with larger batch sizes, the model's performance decreases exponentially. Thus, we have a tradeoff between training time and model performance — to produce a more efficient model, the model's predictive capability must take a hit.

4.6.2 Number of Epochs

Similar to batch size, we can plot training time, training loss, and test accuracy.



We notice a relatively linear relationship between training time and number of epochs, which is to be expected given that each epoch is an extra iteration through the entire training dataset.

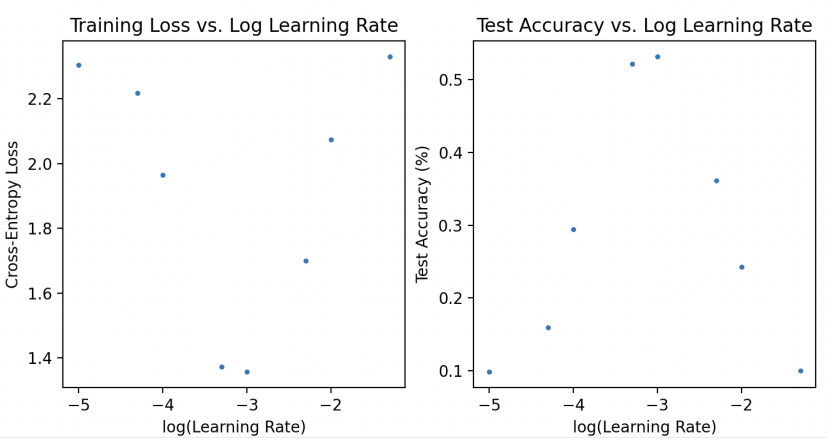
We also observe that the training loss generally improves with a higher number of epochs, but from the test accuracy plot, the model seems to be overfitting for values of the number of epochs greater than 6. The model performance on the test data barely improves and sometimes even regresses with more epochs, which tells us that it is not always optimal to run through the training data more times than less.

4.6.3 Learning Rate

Let us first discuss the learning rate itself. The **learning rate** is a tuning parameter that dictates the how much is being learned at each step of the iteration. More formally, for a step-based learning rate η_n at step n with some decay value $d < 1$, we have

$$\eta_n = \eta_0 d^{\lfloor \frac{1+n}{r} \rfloor}.$$

For varying learning rates in the interval $[10^{-5}, 5 \cdot 10^{-2}]$, we plot the training loss and test accuracy.

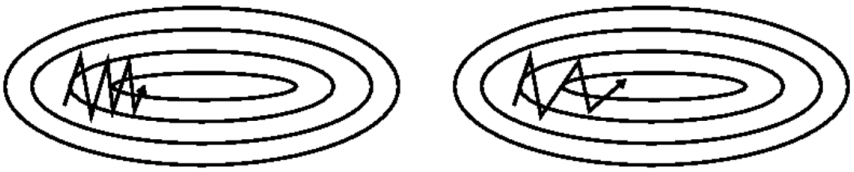


Interestingly, we notice a parabolic pattern for both plots. The training loss and test accuracy both improve with increasing learning rates up until some optimal learning rate, then performance begins to fall. This can be attributed to using too fast of a learning rate, which can cause performance to decrease if parameter updates are too drastic. There obviously is no one-choice-fits-all answer for what is the optimal learning rate for every CNN or machine learning model in general, but we can see the general trend of performance through analyzing the CIFAR-10 example.

4.6.4 Optimizers: Momentum SGD, Adam, RMSProp

We now analyze a selection of a subset of the possible optimizers to use for the learning step in our CNN. The goal is to minimize the loss function provided some multi-dimensional inputs, which is a step-by-step process for updating the model weights to move in the direction of the optimal minimum.

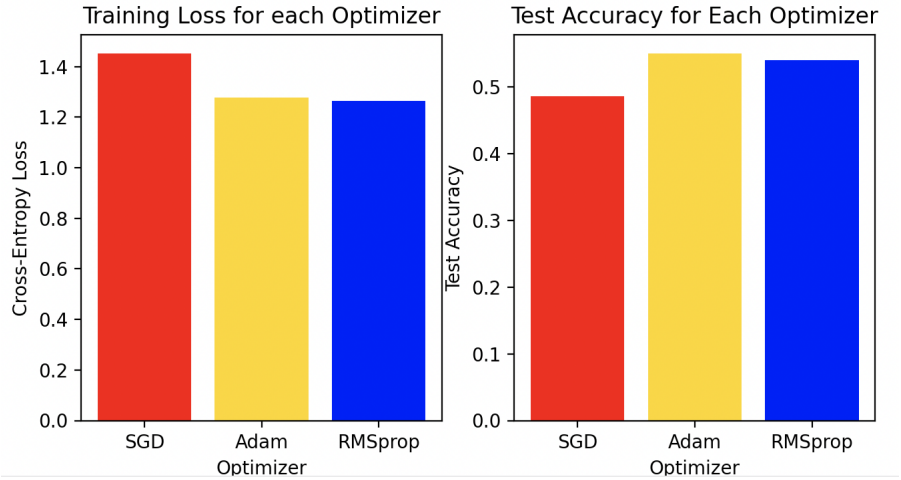
Starting with **momentum stochastic gradient descent (SGD)** is a step-wise process that directs the gradient toward the optimal minimum through model parameter updates. However, SGD with momentum pushes the gradient faster in the right direction than normal SGD. Below is a visualization of this process.



Left: vanilla SGD; Right: SGD with momentum.

Adam is an alternative to SGD with momentum, which is commonly described as combining the advantages of the adaptive gradient (AdaGrad) and root-mean-square propagation (RMSprop) algorithms. This is because it makes use of both the first and second moments of the gradients (both the mean and the variance).

Finally, **RMSprop** is yet another alternative choice. Living up to its name, the algorithm takes the square root of the average of the gradients at each step as its iterative updating process.



We can see that the performance of Adam and RMSprop are relatively similar, but SGD performs slightly worse. The results here are an example of why SGD is becoming a rather outdated optimizer of choice. The reason is because SGD is less capable of handling **saddle points**, points for which the first derivative is equal to 0 but are not a local minima or maxima. If SGD encounters a saddle point, it may get stuck on it and not find a true local minima for the loss function, thus leading to suboptimal results. However, this is not to discount SGD as a legitimate choice for a model, as it is still widely used and relatively simple to understand.

Of course, the optimizers we explored in this tutorial are not a comprehensive list of a noteworthy optimizers, but the comparison of these more popular optimizers can be incredibly helpful. As a side note, the typical choice in present-day study is to use Adam due to its overall reliability in handling saddle points and producing results with reasonable efficiency.

Results and Conclusions

In this section, we covered how to code a CNN in Python with PyTorch and discussed the components of the neural network, including convolutional, pooling, and linear layers for the model architecture, as well as how to train and test the model. We further determined how to select optimal hyperparameters for a given dataset and model by identifying patterns in model performance and training time. Specifically, there is indeed a time-performance tradeoff for setting the batch size. Furthermore, for the number of epochs or learning rate, there are certain optimal choices for each. However, increasing the number of epochs too high can lead to overfitting, and increasing the learning rate too high can lead to decreased performance with training loss and test accuracy. In the next section, we seek to compare our results from the SVM model with our newly constructed CNN.

5 SVM and CNN Comparison on Real-World Data

In conclusion, our CNN model outperformed the SVM model. There are many reasons that this was both expected and unexpected. When comparing SVMs and CNNs in the realm of real-world image classification, we observe key differences in their approaches and performance characteristics. SVMs, being conventional machine learning models, rely on predetermined features and a margin-based classification strategy. They often require manual feature engineering, which may not fully capture intricate patterns and complex structures within images.

Contrastingly, CNNs excel in image classification due to their inherent capacity to automatically learn hierarchical features directly from the data. Leveraging convolutional layers, CNNs autonomously extract local patterns, enabling them to discern intricate structures and relationships within images. This self-learned feature representation, along with the exploitation of spatial hierarchies through multiple layers, positions CNNs as potent tools for image classification.

The comparative advantage of CNNs over SVMs lies in their ability to capture statistical patterns, spatial dependencies, and non-linear relationships within images. The automatic feature learning of CNNs make them particularly effective for tasks involving diverse, large-scale image datasets encountered in real-world scenarios. Thus, CNNs often outperform traditional SVM approaches in real-world image classification.

However, there are trade-offs that come with the performance of CNNs. Compared to SVMs, CNNs are considerably less interpretable because of the infinitesimal modifications that can be made to architecture and hyperparameters, as well as the hierarchical nature of their decision making in comparison to the linear decision boundary or the kernel-induced decision function of SVMs. Furthermore, CNNs are much more computationally intensive than SVMs, and depending on the size and complexity of the dataset may require additional

power sources. As a result of this, they can also be slower in classifying images than SVMs. Based on our own experiments, there was also a significant difference in the amount of time each method took on our real-world dataset, from seconds to minutes for the SVM and CNN, respectively, showing the trade-off for computational efficiency and model performance.

6 Contributions

Masha Zaitsev - Individually worked on all of the 'Exploration of Convolutional Neural Network' section except for the CNN simulated data example. For the simulated data example, I built off of parts of the code written by Jimi and then analyzed the performance of the CNN model and added descriptions. Also contributed to the conclusion.
Jimi Abbott - Individually worked on all of the section "An Exploration of Support Vector Machines". Built code for simulated data in the "An Exploration of Convolutional Neural Networks" section.
Alan Liu - Individually worked on the code and content of the "Experimenting with Convolutional Neural Networks Using CIFAR-10" for CNNs, including analysis of neural network hyperparameters. Also added to the conclusion for the comparison of SVMs and CNNs.
Aditi Raju - Individually wrote the code and contents of the "Experimenting with CIFAR-10 and Support Vector Machines" section. I also wrote the conclusions of the project in the "SVM and CNN Comparison" section.

7 Sources

[1] James, Gareth, et al. *An Introduction to Statistical Learning: With Applications in R.* Springer, 2022.
[2] Krizhevsky, Alex (2009). *Learning multiple layers of features from tiny images. (CIFAR10)*
[3] Nguyen, Son. "A Gentle Explanation of Backpropagation in Convolutional Neural Network (CNN)." *Medium*, Medium, 28 Feb. 2020, medium.com/@ngocson2vn/a-gentle-explanation-of-backpropagation-in-convolutional-neural-network-cnn-1a70abff508b.
[4] Skalski, Piotr. "Gentle Dive into Math behind Convolutional Neural Networks." *Medium*, Towards Data Science, 14 Apr. 2019, towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9.
[5] Bushaev, Vitaly. "Stochastic Gradient Descent with Momentum." Medium, Towards Data Science, 5 Dec. 2017, towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d.
[6] "Training a Classifier." Training a Classifier - PyTorch Tutorials 2.2.0+cu121 Documentation, pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.