MATHEMATICAL ANALYSIS OF CONVOLUTIONAL NEURAL NETWORKS

By

Daniel McCarter

B.S., Albright College, 2020

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of Master of
Science

Department of Mathematical Sciences

Mathematics Program
In the Graduate School
The University of South Dakota
May 2023

The members of the Committee appointed to examine
the Thesis of Daniel McCarter
find it satisfactory and recommend that it be accepted.

DocuSigned by:
Gabriel Picioroaga
299CBE727804473...
____
Chairperson

DocuSigned by:
Nan Jiang
788849FA8F804A0...
____

DocuSigned by:
KC Santosh
BBEF270EA578492...
____

ABSTRACT

In this thesis, the main topic is convolution as a mathematical operation and Convolutional Neural Networks (CNN's). While convolution is classically defined as a function, it can also be defined as an operator from $L^p(\mathbb{R})$ to itself for $1 \leq p \leq 2$ where $T_w(f) = f * w$ given some $w \in L^1(\mathbb{R})$. CNN's use convolution in its convolutional layers. Defining a neural network to be the composition of layer maps, we find that the neural network is, by necessity, Lipschitz. While CNN's can be very powerful for image classification, slight changes to an image can completely fool the network. By augmenting our training data with these modifications, the network's ability to correctly classify images with these modifications significantly increases.

Thesis Advisor G. Picioroaga
G.Picioroaga (Apr 26, 2023 14:59 CDT)
Gabriel Picioroaga

Contents

List of Figures

Chapter 1

Introduction

Convolutional Neural Networks (CNN's) have become popular in the last decade for their applications to image classification and signal processing. Their architecture is well-designed for processing input data such as pixels, images, and audio signals due to the mathematical operation behind it: convolution. While convolution is classically studied in the continuous case, it can be defined for discrete inputs. This makes it perfect for working with images. CNN's can be quite a powerful and effective tool for image classification. However, this is not always the case. Modifying an image even slightly can completely fool the network. How can we fix this?

In this thesis, we present a rigorous study of both convolution in both the continuous and discrete cases and of CNN's. We begin in Chapter 2 by discussing spaces of signals, in particular Hilbert spaces. We then move onto properties of operators and define several operators that can interact with convolution. In Chapter 3, we formally define convolution in the continuous and discrete cases. We extract several key properties regarding norms and then move on to defining convolution as an operator. We then extract more properties and then conclude the chapter by looking at the interaction between the convolution operator and other operators.

We then turn our attention to the structure and training of CNN's in Chapter 4. We discuss the architecture and training of CNN's, as well as the advantages and disadvantages of using them. We then formally define several key components of the network, including the network itself before proving that the network satisfies the Lipschitz property under certain conditions. In Chapter 5, we prepare to actually use the network. We introduce the modifications we will apply to various images, including a blur, a rotation, and a translation into a larger image. We then introduce a technique that will give us insight into how the network approaches classifying a particular image called a GradCAM.

In the final chapter, we run experiments to see how modifications to images affect their classification. We first run these images on both a pre-trained network and a manually trained network. We then run these images on a network whose training data has been augmented to help it correctly classify images with modifications on it. We find that, in each case, augmenting the training data results in an improvement in performance.

Chapter 2

Preliminaries

In this chapter, we will discuss theoretical concepts required for our study of convolution and neural networks. We start by describing and defining different spaces that we will use during the construction of the convolution operator. The spaces and their corresponding norms we will work with are:

$L^2([-\pi,\pi]) = \left\{ f : [-\pi,\pi] \to \mathbb{C} | \int_{-\pi}^{\pi} |f(x)|^2 dx < \infty \right\}$
with norm $\|f\|_2 = \sqrt{\int_{-\pi}^{\pi} |f(x)|^2 dx}$
$L^1(\mathbb{R}) = \left\{ f : \mathbb{R} \to \mathbb{C} | \int_{-\infty}^{\infty} |f(x)| dx < \infty \right\}$ with norm $\|f\|_1 = \int_{-\infty}^{\infty} |f(x)| dx$
$L^2(\mathbb{R}) = \left\{ f : \mathbb{R} \to \mathbb{C} | \int_{-\infty}^{\infty} |f(x)|^2 < \infty \right\}$ with norm $\|f\|_2 = \sqrt{\int_{-\infty}^{\infty} |f(x)|^2 dx}$

We will also consider the discrete versions of the spaces above:

$l^2(\mathbb{Z}) = \left\{ f : \mathbb{Z} \to \mathbb{C} | \sum_{n\in\mathbb{Z}} |f(n)|^2 < \infty \right\}$ with norm $\|f\|_2 = \sqrt{\sum_{n\in\mathbb{Z}} |f(n)|^2}$
$l^1(S) = \{ f : S \to \mathbb{C} | \sum_{x\in S} |f(x)| < \infty \}$ with norm $\|f\|_1 = \sum_{x\in S} |f(x)|$
where S is a countable set
$l^2(S) = \{ f : S \to \mathbb{C} | \sum_{x\in S} |f(x)|^2 < \infty \}$ where S is a countable set, and
$\mathbb{C}^{|S|} = \{ f : S \to \mathbb{C} \}$

If S is finite then $l^2(S) = \mathbb{C}^{|S|}$.

2.1 Properties of Spaces

We now discuss properties of spaces. In particular, we will consider Hilbert spaces and their consequences.

Definition 2.1.1. *Let H be a linear space. We call a function $\langle \cdot, \cdot \rangle : H \times H \to \mathbb{R}$ an inner product on H if $\forall x_1, x_2, x, y \in H$ and $\alpha, \beta \in \mathbb{R}$, we have:*

(i) $\langle \alpha x_1 + \beta x_2, y \rangle = \alpha \langle x_1, y \rangle + \beta \langle x_2, y \rangle$,

(ii) $\langle x, y \rangle = \langle y, x \rangle$, *and*

(iii) $\langle x, x \rangle > 0$ *if* $x \neq 0$.

Definition 2.1.2. *If H is a linear space and is equipped with an inner product on H, then we call H an inner product space with a norm induced by the inner product* $\|f\| = \sqrt{\langle f, f \rangle}$

Definition 2.1.3. *If H is a Banach (complete) space with respect to the norm induced by the inner product, the we call H a Hilbert space.*

It is known that L^2 and l^2 are both Hilbert spaces. Let $f, g \in L^2$, then $\langle f, g \rangle = \int f(x)\bar{g}(x)dx \in L^1$ by the Cauchy-Schwartz Inequality [2].

Theorem 2.1.1. $L^2([-\pi, \pi])$ *admits an orthonormal basis.*

Proof. We first need to show that each vector has a unit norm. Let $e_n \in L^2([-\pi, \pi])$,

where $e_n(t) = \dfrac{e^{int}}{2\pi} = [\cos(nt) + i\sin(nt)] \cdot \dfrac{1}{2\pi}$

The L^2 norm of e_n gives us: $\|e_n\|_2^2 = \displaystyle\int_{-\pi}^{\pi} \sqrt{\frac{\cos^2(t)}{4\pi^2} + \frac{\sin^2(t)}{4\pi^2}} = 1$

Therefore each vector has a unit norm. We now need to show that:

$$\langle e_n, e_m \rangle = \begin{cases} 1 & n = m \\ 0 & n \neq m \end{cases}$$

Case 1: n = m

$$\langle e_n, e_n \rangle = \int_{-\pi}^{\pi} e_n \cdot \bar{e_n} = \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{int} \cdot e^{-int} dt$$

$$= \frac{1}{2\pi} \int_{-\pi}^{\pi} [\cos(nt) + i\sin(nt)] \cdot [\cos(-nt) + i\sin(-nt)] dt$$

$$= \frac{1}{2\pi} \int_{-\pi}^{\pi} [\cos(nt) + i\sin(nt)] \cdot [\cos(nt) - i\sin(nt)] dt$$

$$= \frac{1}{2\pi} \int_{-\pi}^{\pi} \cos^2(nt) + \sin^2(nt) dt$$

$$= \frac{1}{2\pi} \int_{-\pi}^{\pi} dt = \frac{1}{2\pi} t \Big|_{-\pi}^{\pi} = 1$$

Case 2: n ≠ m

$$\langle e_n, e_m \rangle = \int_{-\pi}^{\pi} e_n \cdot \bar{e_m} dt = \frac{1}{2\pi} \int_{-\pi}^{\pi} e^{int} \cdot e^{-imt} dt$$

$$= \frac{1}{2\pi} \int_{-\pi}^{\pi} [\cos(nt) + i\sin(nt)] \cdot [\cos(mt) - i\sin(mt)] dt$$

$$= \frac{1}{2\pi} \int_{-\pi}^{\pi} \cos[(n-m)t] + i\sin[(n-m)t] dt$$

$$= \frac{1}{2\pi(n-m)} \sin[(n-m)t] \Big|_{-\pi}^{\pi} - \frac{i}{2\pi(n-m)} \cos[(n-m)t] \Big|_{-\pi}^{\pi} = 0 \qquad \square$$

So $(e_n)_{n\in\mathbb{Z}}$ is an orthonormal system. It is a known result from Real Analysis that $L^2(E)$ is complete. Although it will not be proven here, this is formally known as the Riesz-Fischer Theorem [2]. Before we move to the next theorem, an important result is presented. This will be used in the proof of the next theorem.

Lemma 2.1.1 (Parseval's Identities). *Let $\{\varphi_n\}$ be an orthonormal basis for a*

Hilbert space H. Then $\forall h, u, v \in H$:

(i) $\|h\|^2 = \sum_{n=1}^{\infty} \langle \varphi_n, h \rangle^2$ *and*

(ii) $\langle u, v \rangle = \sum_{n=1}^{\infty} a_n \cdot b_n$ *where* $\forall n \in \mathbb{N}$, $a_n = \langle u, \varphi_n \rangle$ *and* $b_n = \langle v, \varphi_n \rangle$

We are now ready to discuss an operator from L^2 to l^2 defined by an inner product and extract several key properties:

Theorem 2.1.2. *Let $f \in L^2([-\pi, \pi])$. Define $T : L^2([-\pi, \pi]) \to l^2(\mathbb{Z})$ by*

$T(f)(n) = \langle f, e_n \rangle \quad \forall n \in \mathbb{N}, \quad \forall f \in L^2([-\pi, \pi])$

Then T satisfies the following properties:

(i) T is linear

(ii) T is an isometry

Proof. (i) Let α, β be scalar constants. Then:

$T(\alpha f + \beta g) = \langle \alpha f + \beta g, e_n \rangle$

$= \langle \alpha f, e_n \rangle + \langle \beta g, e_n \rangle$

$= \alpha \langle f, e_n \rangle + \beta \langle g, e_n \rangle$

$= \alpha T(f)(n) + \beta T(g)(n)$

(ii) We want to show $\|T(f) - T(g)\|_{l^2} = \|f - g\|_{L^2}$

Since T is linear, $\|T(f) - T(g)\|_{l^2}^2 = \|T(f - g)\|_{l^2}^2$

By Parseval's Identities, $\quad \|T(f - g)\|_{l^2}^2 = \sum_{n=1}^{\infty} \langle f - g, e_n \rangle^2$

$= \int_{-\pi}^{\pi} |(f - g)|^2 = \|f - g\|_{L^2}^2$

Therefore $\|T(f) - T(g)\|_{l^2} = \|f - g\|_{L^2}$ $\square$

2.2 Operators

In preparation for discussing the convolution operator, we will now discuss several operators related to actions we will later perform on images. We will consider these operators defined over Hilbert spaces.

Definition 2.2.1. *If H is a Hilbert space then $T : H \to H$ is a bounded operator if T is linear and continuous.*

Definition 2.2.2. *Given $a \in \mathbb{R}$, the translation operator $A_a : L^{1,2}(\mathbb{R}) \to L^{1,2}(\mathbb{R})$ is defined by $A_a(f(x)) = f(x-a)$ where either L^1 or L^2 can be used.*

Proposition 2.2.1. *T is a bounded operator on H iff T is linear and $\|T\| < \infty$ where $\|T\| = \sup\{\|T(v)\| \mid \|v\| \leq 1\}$*

Proof. $\boxed{\Rightarrow}$ Let T be bounded on H. By definition, T is both linear and continuous.

Assume $v_n \to v$ in H.

Then $v_n - v \to 0$.

Since T is continuous, we have $T(v_n - v) \to T(0) = 0$.

Therefore T is uniformly continuous and we have that:

$\sup\{\|T(v)\| \mid \|v\| \leq 1\} < \infty$

Thus $\|T\| < \infty$

$\boxed{\Leftarrow}$ Let T be linear and $\|T\| < \infty$. We need to show that T is continuous.

Let $\varepsilon > 0$ and take $\delta = \frac{1}{\|T\|} \cdot \varepsilon$.

Assume $\|v_1 - v_2\| < \delta$ for $v_1, v_2 \in H$.

Then, by using linearity and the Cauchy-Schwartz Inequality, we obtain:

$\|T(v_1) - T(v_2)\| = \|T(v_1 - v_2)\|$

$\leq \|T\| \cdot \|v_1 - v_2\|$

$< \|T\| \cdot \frac{1}{\|T\|} \cdot \varepsilon = \varepsilon.$

Therefore T is continuous and linear, so by definition T is bounded. □

We will now define an adjoint and prove an important theorem that will help us once we move to the convolution operator.

Definition 2.2.3. *Let H be a separable Hilbert space and $T : H \to H$ be a bounded operator. Then an adjoint $T^* : H' \to H$ is called the adjoint of T if $\forall v \in H$ and $w \in H'$ we have the following:*

$$\langle T^*(w), v\rangle_H = \langle w, T(v)\rangle_H$$

For the translation operator $A_a : L^2(\mathbb{R}) \to L^2(\mathbb{R})$, we have:

$\|A_a\|^2 = \sup \|A_a(f)\|_2 = 1$

Theorem 2.2.1. *$A_a^*(f)(x) = f(x + a)$ is the adjoint operator of A_a.*

Proof. We need to show $\langle f, A_a g\rangle_{L^2} = \langle A_a^* f, g\rangle_{L^2}$. Starting with the left side, we get:

$$\langle f, A_a g\rangle_{L^2} = \int_{\mathbb{R}} f(x)\cdot (A_a g)(x)dx$$

$$=\int_{\mathbb{R}} f(x)\cdot g(x-a)dx$$

$$=\int_{\mathbb{R}} f(y)\cdot g(y-a)dy \qquad \text{Substituting } y = x+a \text{ yields:}$$

$$=\int_{\mathbb{R}} f(x+a)\cdot g(x)dx$$

$$=\int_{\mathbb{R}} A_a^* f(x)\cdot g(x)dx$$

$$=\langle A_a^* f, g\rangle_{L^2}. \qquad \square$$

To finish the chapter, we will define another operator related to actions performed on images and its adjoint:

Definition 2.2.4. *The rotation operator $R_\theta : L^{1,2}(\mathbb{R}) \to L^{1,2}(\mathbb{R})$ is defined by $R_\theta(f)(x,y) = f(x', y')$ where $x' = x\cos(\theta) + y\sin(\theta)$ and $y' = y\cos(\theta) - x\sin(\theta)$*

Theorem 2.2.2. *$R^*_\theta(f)(x,y) = f(x'', y'')$ where $x'' = \dfrac{x - y\cos(\theta)}{\cos(\theta)}$ and $y'' = \dfrac{y + x\sin(\theta)}{\cos(\theta)}$ is the adjoint operator of R_θ*

Proof. We need to show that $\langle f, R_\theta\rangle_H = \langle R^*_\theta f, g\rangle_H$. We obtain:

$$\langle f, R_\theta g\rangle_H = \int_{\mathbb{R}} f(x,y)\cdot R_\theta(g)(x,y)dxdy$$

$$= \int_{\mathbb{R}} f(x,y)\cdot g(x\cos(\theta) + y\sin(\theta), y\cos(\theta) - x\sin(\theta))dxdy$$

$$= \int_{\mathbb{R}} f\Big(\frac{x' - y\sin(\theta)}{\cos(\theta)}, \frac{y' + x\sin(\theta)}{\cos(\theta)}\Big)\cdot g(x', y')dx'dy'$$

$$= \int_{\mathbb{R}} f\Big(\frac{x - y\sin(\theta)}{\cos(\theta)}, \frac{y + x\sin(\theta)}{\cos(\theta)}\Big)\cdot g(x, y)dxdy$$

$$= \int_{\mathbb{R}} f(x'', y'')\cdot g(x,y)dxdy$$

$$= \int_{\mathbb{R}} R_\theta^* f(x,y) \cdot g(x,y) dx dy$$

$$= \langle R_\theta^* f, g \rangle_H. \qquad \square$$

Chapter 3

Convolution

We are now ready to talk about convolution. In this chapter we will define convolution in both the discrete and continuous cases, extract properties using the convolution operator, and show examples of convolution in action.

3.1 Classical Convolution

We begin with the classically used definition:

Definition 3.1.1. *For $f, w \in L^1(\mathbb{R})$, we define the convolution of f and w as*

$$(f * w)(x) = \int_{\mathbb{R}} f(y)w(x-y)dy.$$

We will now prove several key properties of convolution:

Theorem 3.1.1. *If $f, g \in L^1(\mathbb{R})$, then $(f * w)$ has the following properties:*

(i) $f * w$ *is well-defined*

(ii) $f * w = w * f$

(iii) $\|f * w\|_1 \leq \|f\|_1 \cdot \|w\|_1$

Proof. (i) Let $F(x) = \int_{\mathbb{R}} f(y)w(x-y)dy$ where $F : \mathbb{R} \to \bar{\mathbb{R}}$

Since F is integrable, F is finite for a.e. $x \in \mathbb{R}$.

By Fubini's Theorem [2], $\int_{\mathbb{R}} |F(x)|dx \leq \int_{\mathbb{R}}\int_{\mathbb{R}} |f(x-y)| \cdot |w(y)|dxdy$

$=\int_{\mathbb{R}} \|f\|_1 |w(y)|dy$

$=\|f\|_1 \cdot \|w\|_1 \leq \infty.$

Therefore $F(x) \in L^1(\mathbb{R})$.

(ii) $(f * w)(x) = \int_{\mathbb{R}} f(y)w(x-y)dy$ Substitute $y = x - z$:

$=\int_{\mathbb{R}} f(x-z)w(z)dz = \int_{\mathbb{R}} w(y)f(x-y)dy$

$=(w * f)(x)$

(iii) $\|f * w\|_1 = \int_{\mathbb{R}} |(f * w)(x)|\, dx \leq \int_{\mathbb{R}}\int_{\mathbb{R}} |f(y)w(x-y)|\, dydx$

$=\|f\|_1 \cdot \|w\|_1.$

□

More general than the previous theorem, the following result from [3] is true:

Theorem 3.1.2. *If* $f \in L^1(\mathbb{R}), g \in L^p(\mathbb{R})$ *for* $1 \leq p \leq \infty$*, then* $(f * g) \in L^P(\mathbb{R})$

Proof. Let $h(x) = (f * g)(x)$. Clearly $|h(x)| \leq \int_{\mathbb{R}} |f(y)| \cdot |g(x-y)|dy$.

Using Minkowski's Inequality, we obtain:

$$\left(\int_{\mathbb{R}} |h(x)|^p dx\right)^{\frac{1}{p}} \leq \int_{\mathbb{R}} \left(\int_{\mathbb{R}} |g(x-y)|^p dx\right)^{\frac{1}{p}} |f(y)|dy$$

$= \|f\|_1 \cdot \|g\|_p < \infty.$

Therefore $(f * g) \in L^P(\mathbb{R})$. $\square$

Definition 3.1.2. *Given $f : \mathbb{R} \to \mathbb{R}$ continuously differentiable and $f \in L^1$, the Fourier Transform of f is defined by:*

$$\mathcal{F}[f(t)](x) = \frac{1}{\sqrt{2\pi}} \cdot \int_{\mathbb{R}} f(t)e^{-itx}dt.$$

One of the fundamental properties of the Fourier Transform is that when it is applied to a convolution, the result is the product of the Fourier Transforms of each function:

Theorem 3.1.3. *$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$ where $\mathcal{F}$ is the Fourier Transform.*

While a proof of this theorem is not given here, one can be found in [1].

3.2 Convolution Operator

Having looked at convolution as a function, we now want to look at it as an operator. This will allow us to find connections between convolution and other operators. To begin our discussion, we look at a very important class of functions for defining the convolution operator.

Definition 3.2.1. *The Schwartz Class of functions is denoted by*

$$S(\mathbb{R}) = \{f \in C^\infty(\mathbb{R}) | \sup_{x\in\mathbb{R}} |x^i f^{(j)}(x)| \leq N_{i,j}, N_{i,j} \in \mathbb{N}\}$$

Generally, the Schwartz class of functions on $\mathbb{R}$ can be thought of as the class of continuously differentiable functions whose derivatives decay faster than any polynomial. More importantly for our purposes, the Schwartz class is dense in both $L^1(\mathbb{R})$ and $L^2(\mathbb{R})$. This result can be found in [3]. Therefore, if $f, w \in S(\mathbb{R})$, then $(f * w) \in S(\mathbb{R})$. We can then define $T : S(\mathbb{R}) \to S(\mathbb{R})$ $T(f) = f*w$ where T is well-defined. This admits an extension to $L^2(\mathbb{R})$ which we will formally define as the convolution operator:

Definition 3.2.2. *Given $w \in L^1(\mathbb{R})$, for $1 \leq p \leq 2$ the convolution operator $T_w : L^p(\mathbb{R}) \to L^p(\mathbb{R})$ is defined by $T_w(f) = f * w$ with norm $\|T_w\| = \sup\{\|T_w\|_p | \|f\|_p \leq 1\}$.*

We will now extract several properties from the convolution operator:

Theorem 3.2.1. *The convolution operator has the following properties:*

(i) T_w *is well-defined*

(ii) T_w *is linear*

(iii) $\|T_w\| \leq \|w\|_1$

(iv) T_w *is commutative*

Proof. (i) By Theorem 3.1.1, T_w is well-defined.

(ii) $T_w(\alpha f + \beta g) = (\alpha f + \beta g) * w$

$$=\int_{\mathbb{R}} [\alpha f + \beta g](y) \cdot w(x-y)dy$$

$$=\int_{\mathbb{R}} [\alpha f(y) + \beta g(y)] \cdot w(x-y)dy$$

$$=\int_{\mathbb{R}} \alpha f(y) \cdot w(x-y) + \beta g(y) \cdot w(x-y)dy$$

$$=\alpha \int_{\mathbb{R}} f(y) \cdot w(x-y)dy + \beta \int_{\mathbb{R}} g(y) \cdot w(x-y)dy$$

$$=\alpha T_w(f) + \beta T_w(g)$$

(iii) Using the definition of $\|T_w\|$ and Theorem 3.1.2, we obtain:

$$\|T_w\| = \sup\{\|T_w(f)\|_p | \|f\|_p \leq 1\}$$

$$\leq \sup\{\|f * w\|_p | \|f\|_p \leq 1\}$$

$\leq \sup\{\|f\|_p \cdot \|w\|_1 | \|f\|_p \leq 1\}$

$\leq \|w\|_1.$

(iv) Let T_w and T_g be the convolution operators of w and g, respectively. Then:

$T_w \circ T_g = T_w(T_g(f))$

$=T_w(f * g)$

$=(f * g) * w$

$=(g * f) * w$

$=g * (f * w)$

$=T_g(T_w(f))$

$=T_g \circ T_w.$

□

Definition 3.2.3. *$F : L^p(\mathbb{R}) \to L^p(\mathbb{R})$ is time-invariant if F is linear and $F \circ A_a = A_a \circ F$ where A_a is the translation operator.*

The following Theorem can be found in [1]:

Theorem 3.2.2. *If F is time-invariant on the space of piece-wise continuous functions, then $\exists w \in L^1(\mathbb{R})$ such that $F(f) = f * w$.*

Theorem 3.2.3. *The convolution operator is time-invariant.*

Proof. $T_w \circ A_a(f)(x) = T_w(f)(x - a)$

$$= \int_{\mathbb{R}} f(y)w(x - a - y)dy$$

$= (f * w)(x - a)$

$= A_a(f * w)(x)$

$= A_a \circ T_w(f)(x).$

$\square$

We will now show examples of convolution in the discrete cases:

Definition 3.2.4. *Given $f, w \in l^2(\mathbb{Z})$, we define the discrete convolution of f and w as $(f * w)[n] = \sum_{k=-\infty}^{\infty} f[k]w[n-k]$*

Definition 3.2.5. *Given $f, w \in l^2(\mathbb{Z})$ with 2-D discrete signals, we define the discrete convolution of f and w as $(f * w)[n, m] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i, j]w[n-i, m-j]$*

Instead of using continuous data, we will replace this with finite dimensional data to approximate L^2 or l^2. In the 1-D case, we replace $f(x) \in L^p(\mathbb{R})$ for $1 \leq p \leq 2$ with $[f_i]_{i=1}^k$ where $\|f\|^2 = \sum_{i=1}^{k} |f_i|^2$. This is a suitable interpretation for audio signals or images if we think of image data, which is a matrix, as a large row. In the 2-D case, we replace $f(x) \in L^p(\mathbb{R})$ with $[f(i, j)]_{j=1,\ldots,m}^{i=1,\ldots,n}$. This is a more suitable interpretation for image-like data.

Chapter 4

Structure and Training of Convolutional Neural Networks

Convolutional Neural Networks (CNN's) are a special type of neural network often used for image, audio, and speech recognition. There are three main types of layers: convolutional layers, pooling layers, and the fully connected layer. The convolutional layers are the foundation of the network and is where most of the computing occurs. Inputs are convolved with a filter function to give an output called a feature. This filter has weights on it that are created and adjusted during the training process. This is done until what is called a feature map is extracted. There may be more than one convolutional layer, if so the process is repeated in the next layer. The pooling layers are similar to the convolutional layers in that a filter is applied to each input, but they differ in that the filter no longer has any weights. The filter is an aggregation function, usually either a max or averaging function. While some data is usually lost in the pooling layer, it helps reduce complexity and can prevent potential overfitting. The fully-connected layer has a different use. While the previous two types are used for feature extraction, the fully-connected layer is used for classification. The classification is based off of the filters applied to the features and the resulting feature map. An activation function is used to give a prediction between 0 and 1.

Why are CNN's better equipped for image classification than other types of neural networks? There are three main reasons: sparse interactions, parameter sharing, and equivariant representations. In other neural networks, layers often use matrix multiplication with a parameter matrix that describes the interactions between each input unit and each output unit. This means that there are often a large number of parameters, which can slow the network down. However, this is not the case in CNN's. In CNN's, the filter is typically made smaller than the input. Instead of having the filter detect every interaction between units, the filter can detect more meaningful features such as edges. This means that CNN's will have sparse interactions between units. Since these features might only have a couple hundred pixels instead of millions, we can store significantly fewer parameters. This improves efficiency and reduces the burden of the hardware on the model.

Parameter sharing is when a single parameter is used for more than one function in a model. In other types of neural networks, each element of the parameter matrix is used once to compute a feature and then essentially forgotten as it is not used again. In CNN's each parameter of the filter is used for every input with some exceptions, such as edges. This means that instead of learning a new set of parameters for each input, the network only needs to know one set of parameters for every input. While this does not affect the efficiency of the network, it does reduce the hardware requirements for the model. We can call the number of parameters required k. As discussed in the previous paragraph, k is significantly smaller than m. Since m and n are often the same size, k is so much smaller than $m \times n$, the hardware requirements for CNN's are significantly smaller than for other models.

CNN's are trained using what is called "supervised learning". This consists of two phases: the forward phase and the backward phase. During the forward phase, the network will store any inputs it receives that it'll need for the backward phase. Once the forward phase is completed, the backwards phase will begin. In this phase, each layer will receive a gradient and return a gradient. The gradient it receives is the gradient of loss concerning its outputs, and the gradient it returns is the gradient of loss concerning its inputs. This process is called gradient descent. The objective during training is to minimize the loss function, while simultaneously maximizing the performance of the network. Here is an example of a loss function:

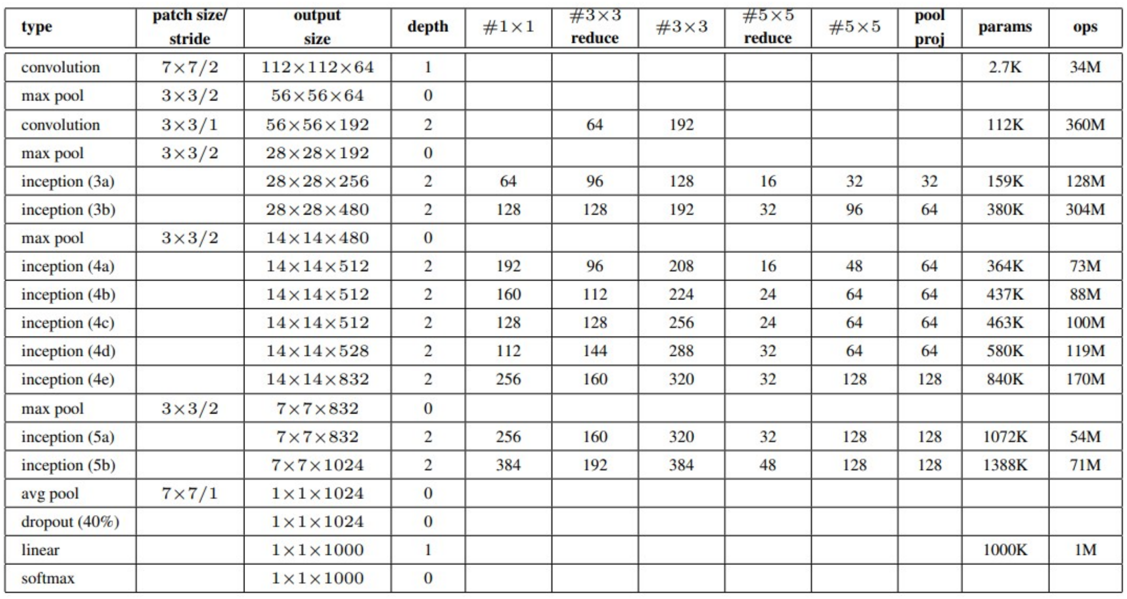
$$f(x) = \sum_{i=1}^{n} \|Ax_i + b - y_i\|^2.$$

This loss function could be used if the training used linear regression. It measures the difference between the predicted value and the actual value. During the backward phase, the loss function is used to adjust the weights in the convolutional layers. The goal is to find A and b such that the loss function is minimized.

There are two major problems that we want to avoid during training: underfitting and overfitting. Underfitting occurs when the network cannot identify dominant trends in the training data so it will perform poorly on both the training data and the testing data. This can be caused by a small training sample or an oversimplified model network. This issue can be solved by either creating a suitable training sample or adding more layers to the network. Since underfitting cause performance issues during the training phase, it can be detected earlier and easier.

Overfitting is almost the exact opposite problem of underfitting. Overfitting occurs when the training error is significantly smaller than the testing error. The network will perform phenomenally on the training data but will struggle with testing data that it hasn't seen yet. This is often the result of a training sample that is too large since the network will become very sensitive to specific parameters of the training data. This can be harder to identify since the network will do very well with the training data and potentially testing data that is very similar to the training data. Diversifying your training sample can help with overfitting.

We will now discuss a particular CNN called GoogLeNet. GoogLeNet is a deep CNN that is often used for image classification, among other things. The size of the network is 224×224 with 22 layers trained on 1000 categories and over one million images. Every convolutional layer uses a ReLU as an activation function, which we formally define and discuss later in the chapter. Here is a table showing the architecture of the network taken from [7]:



type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figure 4.1: Architecture of GoogLeNet

We will now formally define features, layers, and the net as functions:

Definition 4.0.1. *Given a matrix M and a vector $\vec{v}$, we define a feature map $F : \mathbb{R}^n \to \mathbb{R}^m$ as one of the following:*

(i) $F(x) = M \cdot x + \vec{v}$

(ii) $F(x) = M * x$

(iii) $F(x) = \langle M, x \rangle$

In case (iii), M can be replaced by a vector.

Definition 4.0.2. *We define a layer map $L : \mathbb{R}^n \to \mathbb{R}^m$ as*

$L(x) = (\sigma_1(F_1(x)), \sigma_2(F_2(x)), ..., \sigma_m(F_m(x)))$

where σ_i are activation functions and F_i are feature maps.

Definition 3.3: A neural network of depth k is a map $\Phi : \mathbb{R}^n \to \mathbb{R}^m$ where

$\Phi(x) = L_k \circ L_{k-1} \circ ... \circ L_1(x)$

There are several different types of activation functions that can be chosen. In general, non-linear activation functions are preferred. This is because linear activation functions, which can be characterized as $f(x) = ax+b$, do not allow for gradient descent during the backwards training phase, as the derivative of $f(x)$ will be constant. In addition, if a linear activation function is used, then every layer of the network can be written in terms of the first layer, turning our network from a multi-layer network into just a single layer. Therefore, non-linear activation functions are preferred.

A sigmoid function can be chosen as an activation function. We define a sigmoid function, also called a logistic function, as $f(x) = \dfrac{1}{1+e^{-x}}$ with $f'(x) = f(x)f(1 - f(x))$. The range of this function is (0,1) so it can work well for outputting probabilities. Sigmoids are also nice because they are continuously differentiable. Another function that can be chosen as an activation function is the hyperbolic tangent function $f(x) = \tanh x$ with $f'(x) = \text{sech}^2(x)$. This function is similar to the sigmoid function in that its range is (-1,1) and is continuously differentiable.

Both of these functions suffer from the same problem called the "vanishing gradient". The range of both of these function's derivatives is (0,1). During the backwards phase, partial derivatives are computed using the chain rule. Therefore, in a sufficiently large network, when enough of these derivatives are multiplied together the gradient will exponentially decrease. This means that the weights created for each layer during training will not be accurate and will hinder the accuracy of the network.

To solve this problem, a Rectified Linear Unit (ReLU) is often used as an activation function. We define a ReLU as:

$$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

We define the derivative of the ReLU as:

$$f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

Since not every input will be activated, the ReLU is far more efficient compared the the sigmoid or tanh functions. It is also easier to optimize since it behaves as a linear function for positive values while being non-linear for negative values. This disadvantage of a ReLU is that it can suffer from what is called the dying ReLU. Since negative values will make the gradient 0, during the backwards phase weights will not always be updated. This hinders the network's ability to be trained from the data accurately. This problem can be fixed by using a Leaky ReLU function. We define the Leaky ReLU function as:

$$f(x) = \begin{cases} x & x > 0 \\ 0.01x & x \leq 0 \end{cases}$$

We define the derivative of the Leaky ReLU as:

$$f(x) = \begin{cases} 1 & x > 0 \\ 0.01 & x < 0 \end{cases}$$

The Leaky ReLU maintains all advantages of the ReLU while not disrupting the backwards training phase for negative values. If one desires, the value 0.01 can be changed to a different weight that might be more useful for the particular network. This type of ReLU is called a Parametric ReLU.

We will now introduce one well-known property from analysis and relate it to our neural network.

Definition 4.0.3. *: Let f be a real-valued function defined on a set E. f is Lipschitz if $\exists c \geq 0$ such that $\|f(x) - f(y)\| \leq c \cdot \|x - y\|\ \forall x, y \in E$.*

Definition 4.0.4. *We define the Lipschitz constant as*

$c = \inf\{d \mid \|f(x) - f(y)\| \leq d \cdot \|x - y\|\ \forall x, y \in E\}$.

Lemma 4.0.1. *Each feature map is Lipschitz.*

Proof. **Case 1:** $F(x) = M \cdot x + \vec{v}$

Using linearity of multiplication and the Cauchy-Schwartz Inequality, we obtain:

$\|F(x) - F(y)\| = \|M \cdot x + \vec{v} - M \cdot y - \vec{v}\|$

$= \|M \cdot (x - y)\|$

$\leq \|M\| \cdot \|x - y\|$

Case 2: $F(x) = M * x$

Using linearity of convolution and Theorem 2.0.1, we obtain:

$\|F(x) - F(y)\| = \|M * x - M * y\|$

$= \|M * (x - y)\|$

$\leq \|M\| \cdot \|x - y\|$

Case 3: $F(x) = \langle M, x\rangle$

Using linearity of the inner product and the Cauchy-Schwartz Inequality for norms, we obtain:

$\|F(x) - F(y)\| = \|\langle M, x\rangle - \langle M, y\rangle\|$

$= \|\langle M, x - y\rangle\|$

$\leq \|M\| \cdot \|x - y\|$

□

Lemma 4.0.2. *If each activation function σ_i is Lipschitz, then each layer map is Lipschitz.*

Proof. Assume each activation function σ_i is Lipschitz with a corresponding Lipschitz constant c_i and denote the Lipschitz constant for each feature map as d_j. Then:

$$\|L(x) - L(y)\| = \|(\sigma_1(F(x)), ..., \sigma_m(F(x))) - \sigma_1((F(y)), ..., \sigma_M(F(y)))\|$$

$$\leq c_1 \cdot d_1 \|(\sigma_2(F(x)), ..., \sigma_m(F(x))) - (\sigma_2(F(y)), ..., \sigma_m(F(y)))\|$$

.

.

.

$$\leq c_1 \cdot ... \cdot c_m \cdot d_1 \cdot ... \cdot d_m \cdot \|x - y\|$$

□

Theorem 4.0.1. *If each activation function σ_i is Lipschitz, then Φ is also Lipschitz.*

Proof. Let each σ_i be Lipschitz with a corresponding Lipschitz constant c_i and $\Phi(x)$ be a neural network of depth k. Then $\forall x, y \in \mathbb{R}^n$:

$$\|\Phi(x) - \Phi(y)\| = \|L_k(L_{k-1} \circ ... \circ L_1(x)) - L_k(L_{k-1} \circ ... \circ L_1(y))\|$$

$$\leq c_k \cdot \|L_{k-1}(L_{k-2} \circ ... \circ L_1(x)) - L_{k-1}(L_{k-2} \circ ... \circ L_1(y))\|$$

$$\leq c_k c_{k-1} \cdot \|L_{k-2}(L_{k-3} \circ ... \circ L_1(x)) - L_{k-2}(L_{k-3} \circ ... \circ L_1(y))\|$$

.

.

.

$$\leq c_k c_{k-1} c_{k-2} ... c_2 \cdot \|L_1(x) - L_1(y)\|$$

$$\leq c_k c_{k-1} c_{k-2} ... c_2 c_1 \cdot \|x - y\|$$

Taking $c_0 = c_k c_{k-1} c_{k-2} ... c_2 c_1$, we obtain $\|\Phi(x) - \Phi(y)\| \leq c_0 \cdot \|x - y\|$ □

Therefore, given Lipschitz activation functions, our neural network is also Lipschitz. Denoting the Lipschitz constant of Φ as c, we know that $c \leq c_0$. However, we don't know the exact value of this Lipschitz constant. Since c_0 is a product of k distinct values , it can be quite large. Therefore, even though x and y might be close to each other, $\Phi(x)$ and $\Phi(y)$ might not be. If our Lipschitz constant c is small, this means that if there is a small difference in x and y, then there would be a small difference in the output by the network. Large changes in the network would require a large change in x. Therefore, if our network has a small Lipschitz constant, then a very small change to an input such as an image or pixel would only correspond to a very small change in the output by the network.

Chapter 5

Using the Network

We are almost ready to put our network into action. In this chapter, we will discuss the various actions performed on our training data. We will discuss what the network will output and how we will interpret it. We will begin with discussing the blur, rotation, and translation actions on the images in our training data.

To blur an image, we need to use convolution. We convolve the image with a filter matrix with entries summing up to 1 and a filter center of varying magnitude to obtain a blurred version of our original image.

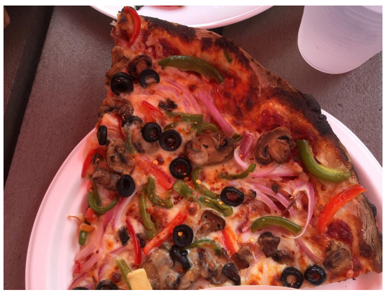


Figure 5.1: Image before Blur

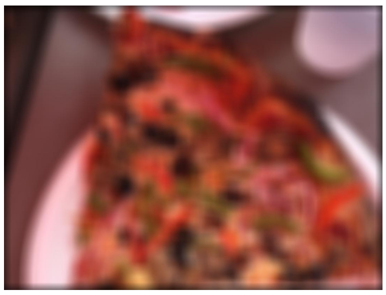


Figure 5.2: Blurred Image

The next action we will discuss is rotation. Rotation is applied by taking the center point of an image and rotating each pixel counterclockwise around it by an angle θ. For this paper, we will let $\theta = 90$ degrees. We then obtain an image that has been rotated 90 degrees counterclockwise:

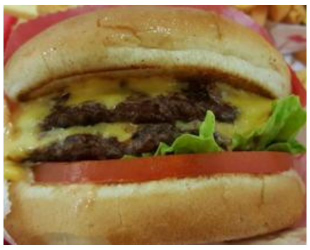


Figure 5.3: Image before Rotation

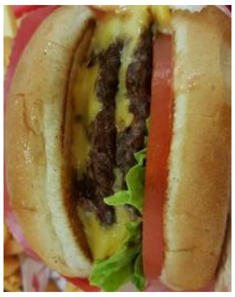


Figure 5.4: Rotated Image

The final action we will perform on our images is translation. This is done by shifting the rows and columns of the image matrix diagonally and assigning zeroes in their place. We then obtain our original image embedded in a larger blank image:

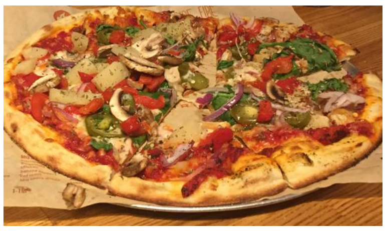


Figure 5.5: Image before Translation

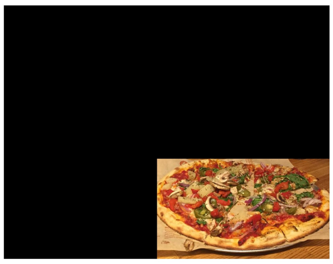
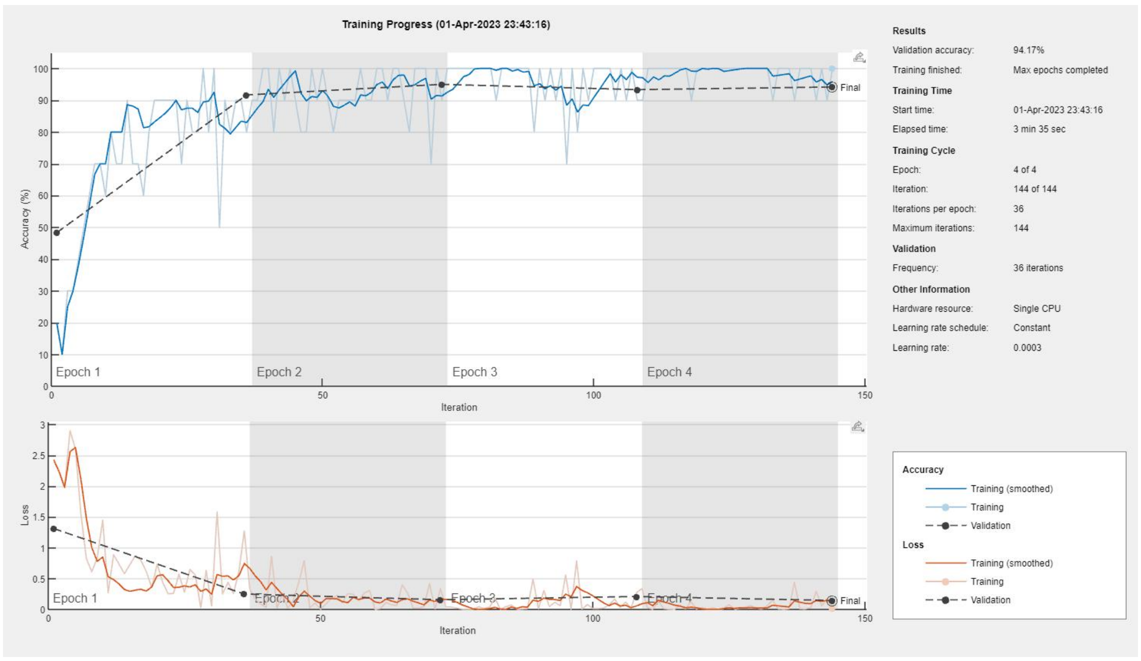


Figure 5.6: Translated Image

We will train our network on different data sets and see how the performance of the network changes. Regardless of which data set we use, once the training phase is complete the network will give us an image of the training progress:

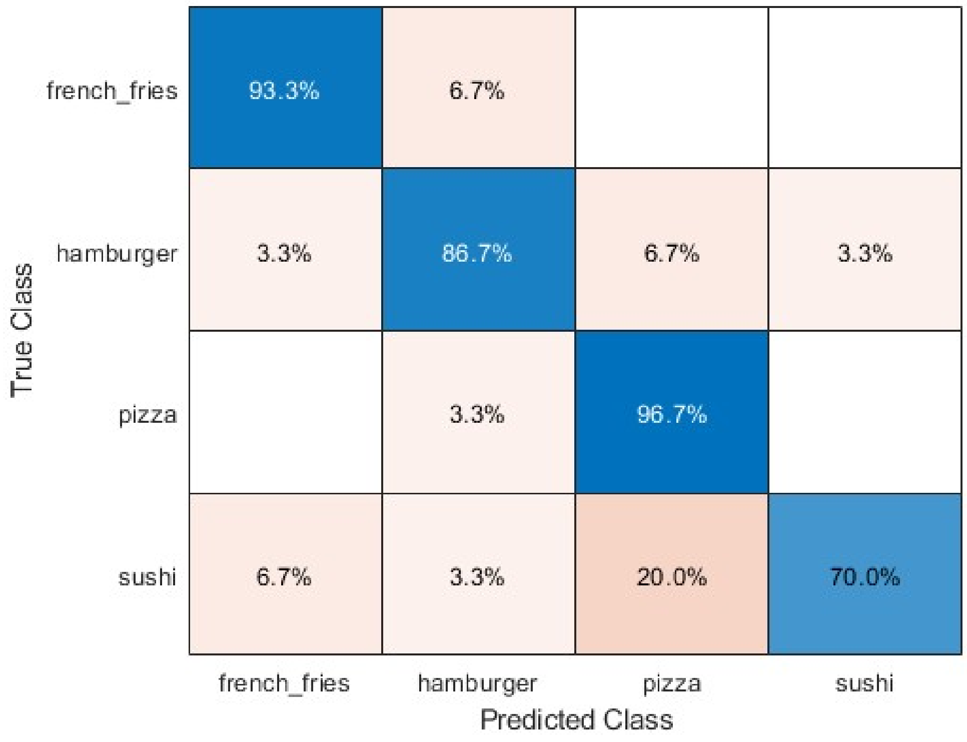


Results

Validation accuracy:	94.17%
Training finished:	Max epochs completed
Training Time	
Start time:	01-Apr-2023 23:43:16
Elapsed time:	3 min 35 sec
Training Cycle	
Epoch:	4 of 4
Iteration:	144 of 144
Iterations per epoch:	36
Maximum iterations:	144
Validation	
Frequency:	36 iterations
Other Information	
Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.0003

Figure 5.7: Summary of Training Progress

Once the training phases are complete, we will be able to obtain a confusion matrix. This matrix will show us how each class is performing and where potential confusion between classes may lie:



	french_fries	hamburger	pizza	sushi
french_fries	93.3%	6.7%		
hamburger	3.3%	86.7%	6.7%	3.3%
pizza		3.3%	96.7%	
sushi	6.7%	3.3%	20.0%	70.0%

Figure 5.8: Example Confusion Matrix

From the matrix, we can see that both the pizza and french fries classes are performing very well, while the hamburger class is performing adequately and the sushi class is struggling to perform accurately. We can also tell what other classes the network is confusing sushi for. This is very useful because we now have some insight into where the network is going wrong.

However, we aren't sure exactly how the network is misclassifying the sushi class. We need to see how the network chooses its classifications. In order to understand this, we will produce a visual called a Gradient-weighted Class Activation Mapping, or Grad-CAM for short. This technique is explicitly described in [8]. This image represents how the network made a decision on classifying a particular image using gradients and produces a visual similar to a heat map, showing where the network is targeting its efforts. We define $\mathcal{L}^c \in \mathbb{R}^{u \times v}$ as the class-discriminative map of width u and height v for a given class c. In order to find $\mathcal{L}^c$, we must compute $\dfrac{\partial y^c}{\partial F^k}$ where y^c is the gradient of the score for the given class c and F^k is the k^{th} feature map of a convolutional layer. The next step is to use Global Average Pooling (GAP) on these gradients. This will average each gradient of each feature map. The advantage of using this technique is that it creates correspondences between the feature maps and the classes. Using this technique on these gradients, we can compute the neuron importance weights:

$$\alpha_k^c = \frac{1}{Z} \sum_{i=1}^{u} \sum_{j=1}^{v} \frac{\partial y^c}{\partial F_{i,j}^k}$$

where Z is the number of feature maps. These weights capture the importance of a feature map k for the class c. Before we can find $\mathcal{L}^c$, we will use a ReLU to act as an activation function for each feature. We can now calculate $\mathcal{L}^c$:

$$\mathcal{L}^c = ReLU\Big(\sum_k \alpha_k^c F^k\Big)$$

This produces a heat map with the same size as the feature maps:

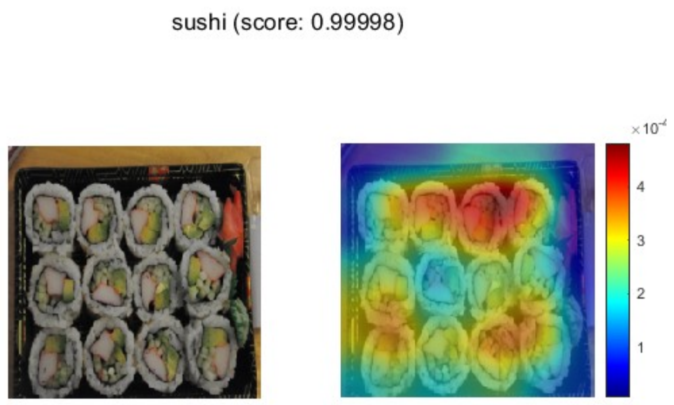


Figure 5.9: Grad-CAM for a correct sushi classification

We can now see what activations will give a sushi classification. We will be able to compare wrong classifications to the correct ones and see where the differences are.

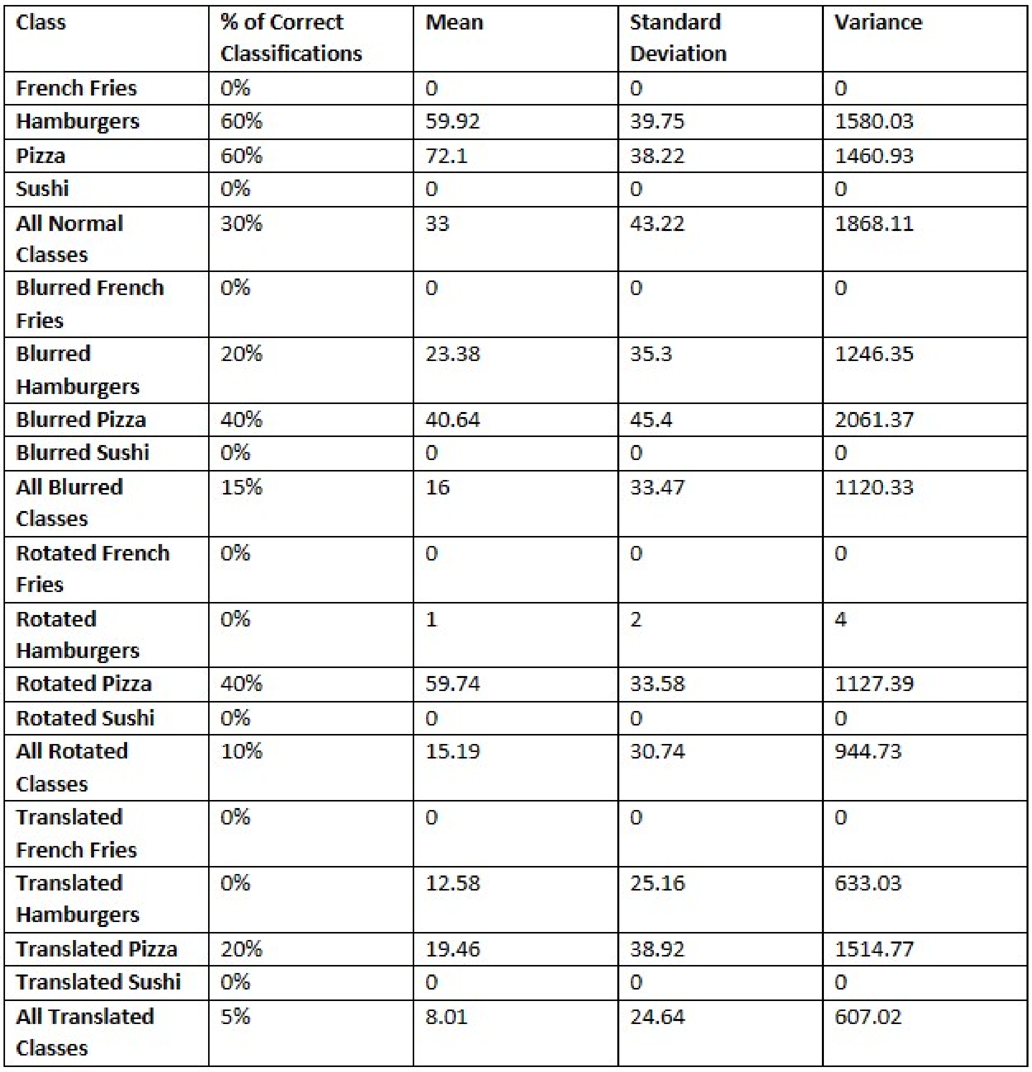
Chapter 6

Results

Our goal for these experiments is to see how the image classification is changed when an image is altered by either a blur, a rotation, or a translation. We have a testing data set of 20 images for each of our four classes: french fries, hamburgers, pizza, and sushi. We will first run these through the GoogLeNet and see how the classification changes. We will then train a network on these four classes with 150 images for each class. We will see how the testing data performs on this network. Finally, we will then run several different targeted trained networks with augmented data for each of the actions and check how much improvement occurred. This will be validated by looking at an independent testing data set and seeing how the network performs.

6.1 GoogLeNet

We ran our independent testing data through the GoogLeNet in four different ways: unchanged, blurred, rotated, and translated. Here are the results:



Class	% of Correct Classifications	Mean	Standard Deviation	Variance
French Fries	0%	0	0	0
Hamburgers	60%	59.92	39.75	1580.03
Pizza	60%	72.1	38.22	1460.93
Sushi	0%	0	0	0
All Normal Classes	30%	33	43.22	1868.11
Blurred French Fries	0%	0	0	0
Blurred Hamburgers	20%	23.38	35.3	1246.35
Blurred Pizza	40%	40.64	45.4	2061.37
Blurred Sushi	0%	0	0	0
All Blurred Classes	15%	16	33.47	1120.33
Rotated French Fries	0%	0	0	0
Rotated Hamburgers	0%	1	2	4
Rotated Pizza	40%	59.74	33.58	1127.39
Rotated Sushi	0%	0	0	0
All Rotated Classes	10%	15.19	30.74	944.73
Translated French Fries	0%	0	0	0
Translated Hamburgers	0%	12.58	25.16	633.03
Translated Pizza	20%	19.46	38.92	1514.77
Translated Sushi	0%	0	0	0
All Translated Classes	5%	8.01	24.64	607.02

Figure 6.1: Testing Statistics for GoogLeNet

As you can see, the GoogLeNet did not perform very well overall on any of the images. However, there is a clear decrease in performance once we alter the images, with the worst cases being the translation and the heavy blur. There are several possible reason for this. The first is that the data that GoogLeNet was trained on was not augmented with altered data. This means that the network hasn't seen these deformities and doesn't know how to respond to them. The second is that there are a lot of categories that it is trained on. Since there are so many categories, it is possible that a blur on an image will make the image resemble a different category. Here is an example of an image that performed well without a blur but failed with one:

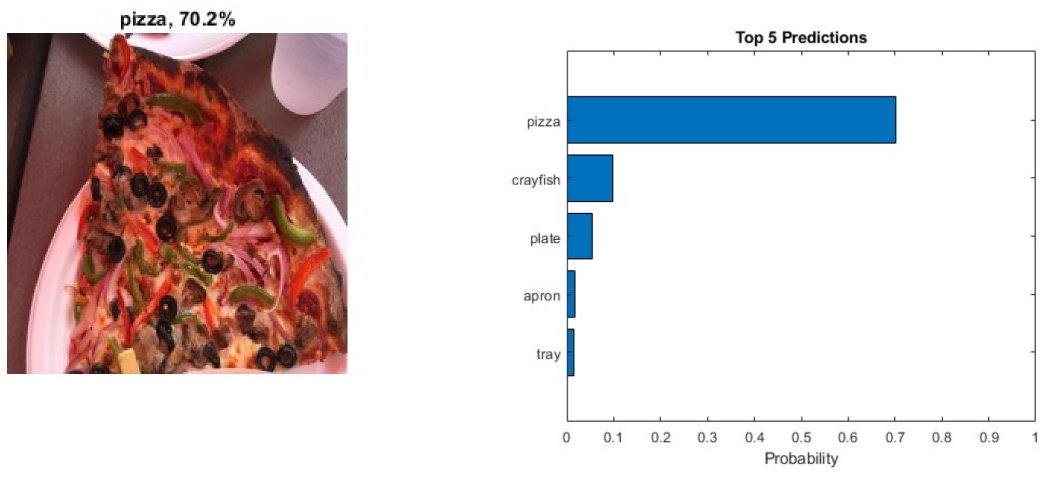


Figure 6.2: GoogLeNet Classification and Prediction for Normal Image

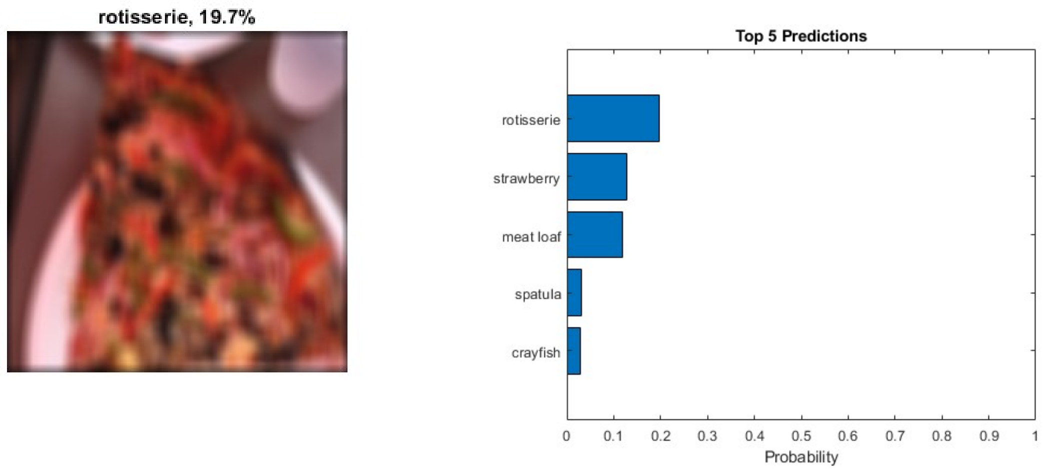
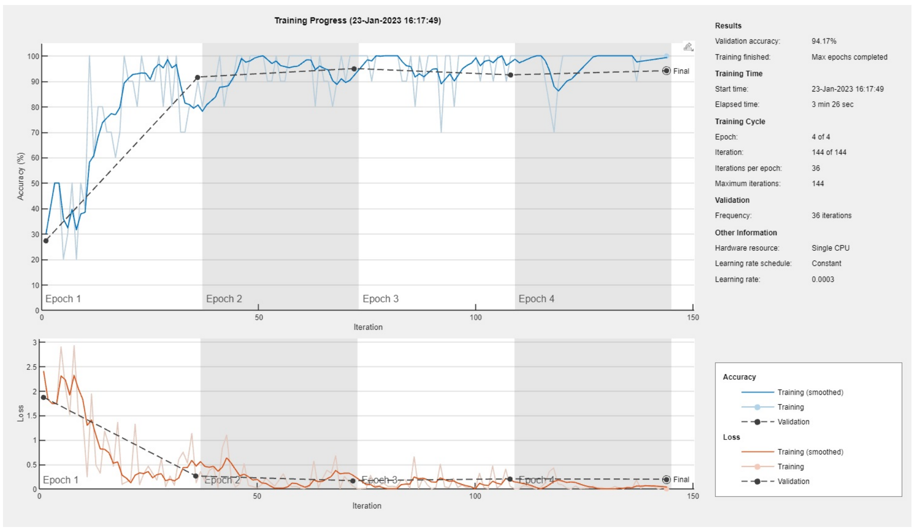


Figure 6.3: GoogLeNet Classification and Prediction for Blurred image

6.2 Trained Network

We will first train our network on a training set with images that have not been changed at all. We will have four different classes: Pizza, Sushi, Hamburger, and French Fries. Each class will have 150 images in the total set, so there will be 90 images in the training set, 30 in the validation set, and 30 in the testing set. We will then run a set of independent data on the network for each class and perform a statistical analysis to determine how well our network is performing. We will compare this data to later networks with different training data. Training on this data set gives us the following training progress:



Results
Validation accuracy: 94.17%
Training finished: Max epochs completed
Training Time
Start time: 23-Jan-2023 16:17:49
Elapsed time: 3 min 26 sec
Training Cycle
Epoch: 4 of 4
Iteration: 144 of 144
Iterations per epoch: 36
Maximum iterations: 144
Validation
Frequency: 36 iterations
Other Information
Hardware resource: Single CPU
Learning rate schedule: Constant
Learning rate: 0.0003

Figure 6.4: Training Progress

The validation is 94% which is not bad. However, the real test will come when we test outside data on the network. We obtain this confusion matrix:

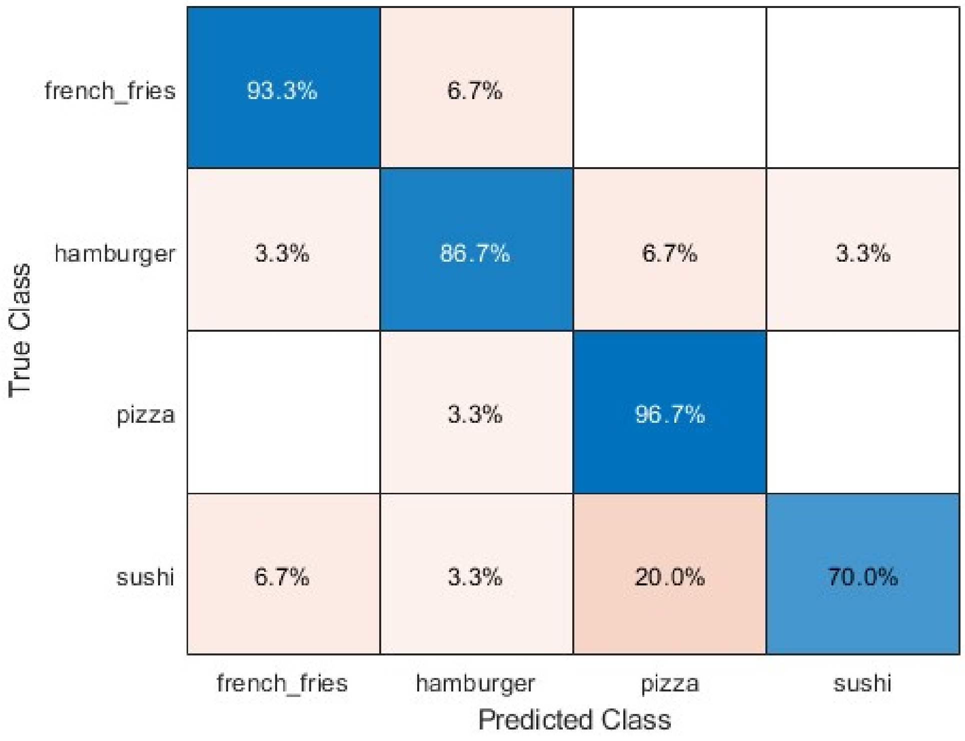
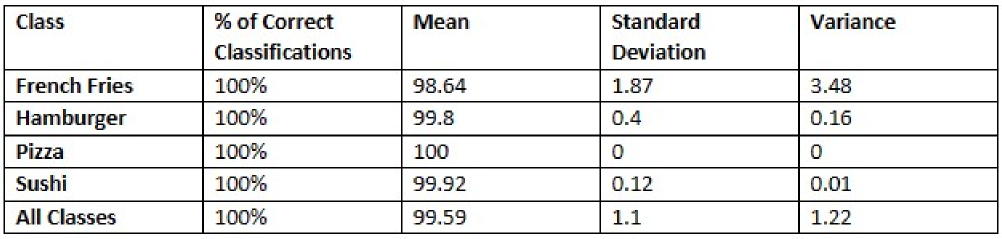


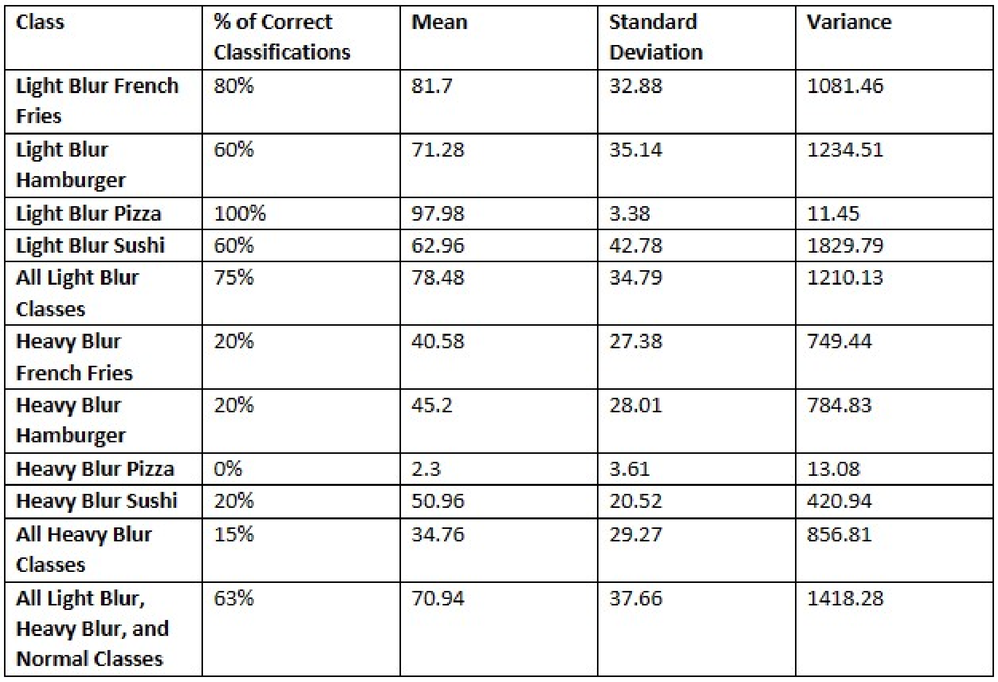
Figure 6.5: Confusion Matrix

We can see that while the pizza, french fries, and hamburger classes performed well, the sushi class barely passed our desired threshold of 70%. We will now run our independent testing data through this network along with the altered versions of the images:



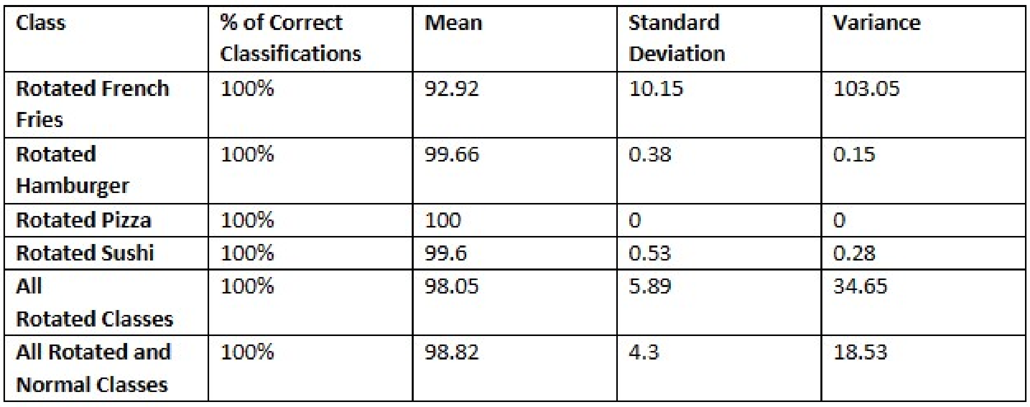
Class	% of Correct Classifications	Mean	Standard Deviation	Variance
French Fries	100%	98.64	1.87	3.48
Hamburger	100%	99.8	0.4	0.16
Pizza	100%	100	0	0
Sushi	100%	99.92	0.12	0.01
All Classes	100%	99.59	1.1	1.22

Figure 6.6: Testing Statistics for Normal Data



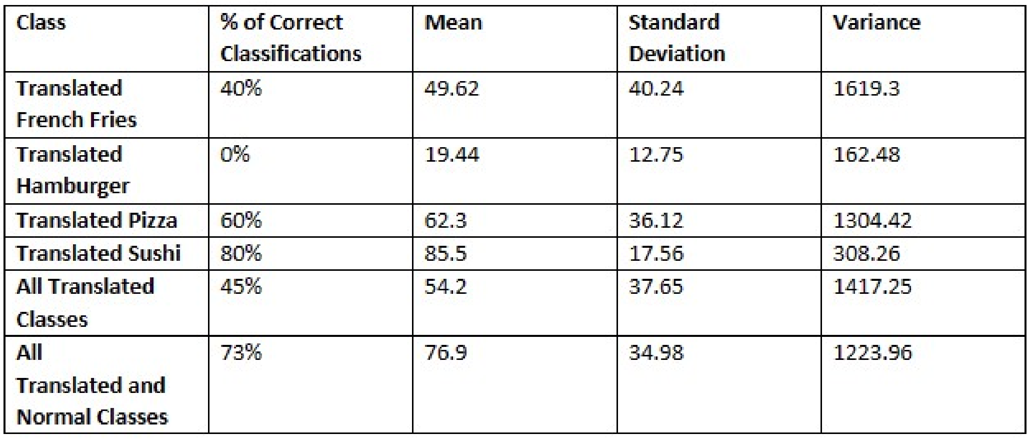
Class	% of Correct Classifications	Mean	Standard Deviation	Variance
Light Blur French Fries	80%	81.7	32.88	1081.46
Light Blur Hamburger	60%	71.28	35.14	1234.51
Light Blur Pizza	100%	97.98	3.38	11.45
Light Blur Sushi	60%	62.96	42.78	1829.79
All Light Blur Classes	75%	78.48	34.79	1210.13
Heavy Blur French Fries	20%	40.58	27.38	749.44
Heavy Blur Hamburger	20%	45.2	28.01	784.83
Heavy Blur Pizza	0%	2.3	3.61	13.08
Heavy Blur Sushi	20%	50.96	20.52	420.94
All Heavy Blur Classes	15%	34.76	29.27	856.81
All Light Blur, Heavy Blur, and Normal Classes	63%	70.94	37.66	1418.28

Figure 6.7: Testing Statistics for Blurred Data



Class	% of Correct Classifications	Mean	Standard Deviation	Variance
Rotated French Fries	100%	92.92	10.15	103.05
Rotated Hamburger	100%	99.66	0.38	0.15
Rotated Pizza	100%	100	0	0
Rotated Sushi	100%	99.6	0.53	0.28
All Rotated Classes	100%	98.05	5.89	34.65
All Rotated and Normal Classes	100%	98.82	4.3	18.53

Figure 6.8: Testing Statistics for Rotated Data



Class	% of Correct Classifications	Mean	Standard Deviation	Variance
Translated French Fries	40%	49.62	40.24	1619.3
Translated Hamburger	0%	19.44	12.75	162.48
Translated Pizza	60%	62.3	36.12	1304.42
Translated Sushi	80%	85.5	17.56	308.26
All Translated Classes	45%	54.2	37.65	1417.25
All Translated and Normal Classes	73%	76.9	34.98	1223.96

Figure 6.9: Testing Statistics for Translated Data

This network does very well with images that have not been altered, images with a slight blur, and rotated images. The network begins to falter with translations but is still able to correctly identify nearly half of the translated independent testing data. However, once a larger blur is applied to the images, the network completely falls apart with only 15% of images being correctly identified. Let us look at an example of the network failing to correctly classify an image once an alteration has been applied:

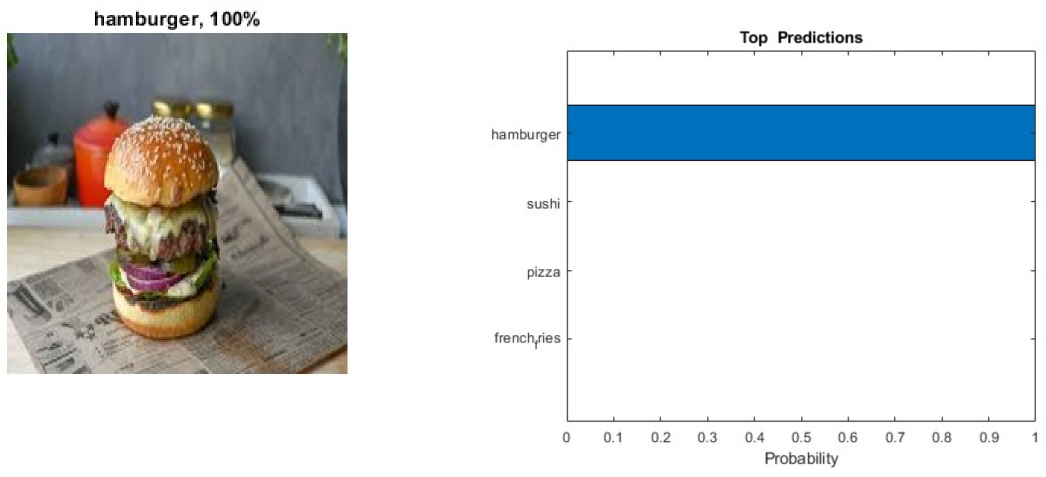


Figure 6.10: Classification and Prediction for Normal Image

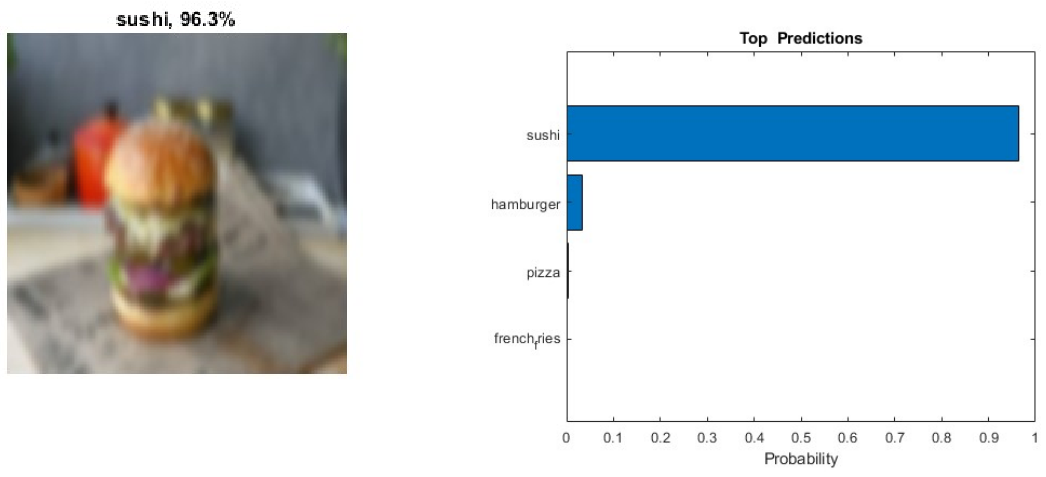


Figure 6.11: Classification and Prediction for Blurred Image

We can also compare the GradCAM for each image:

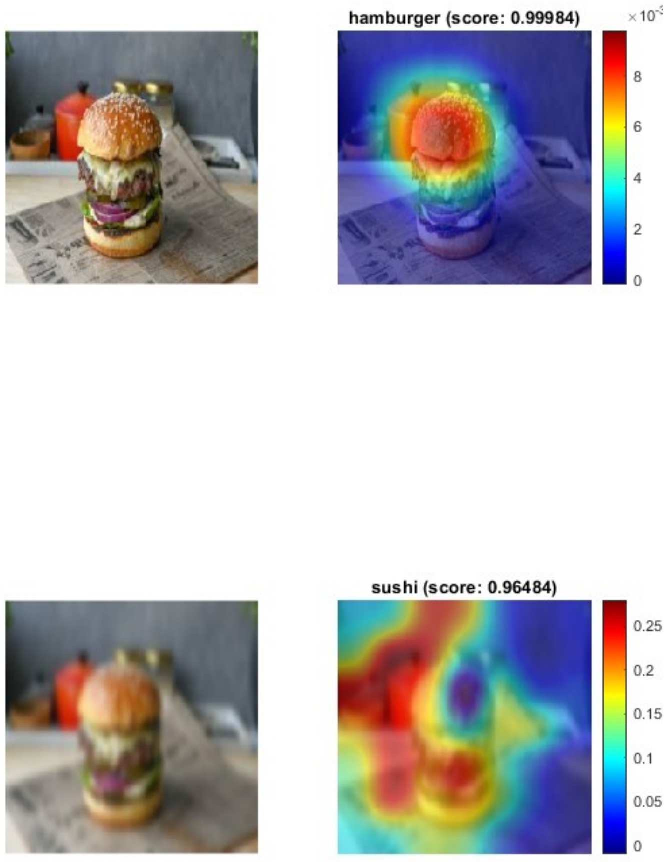
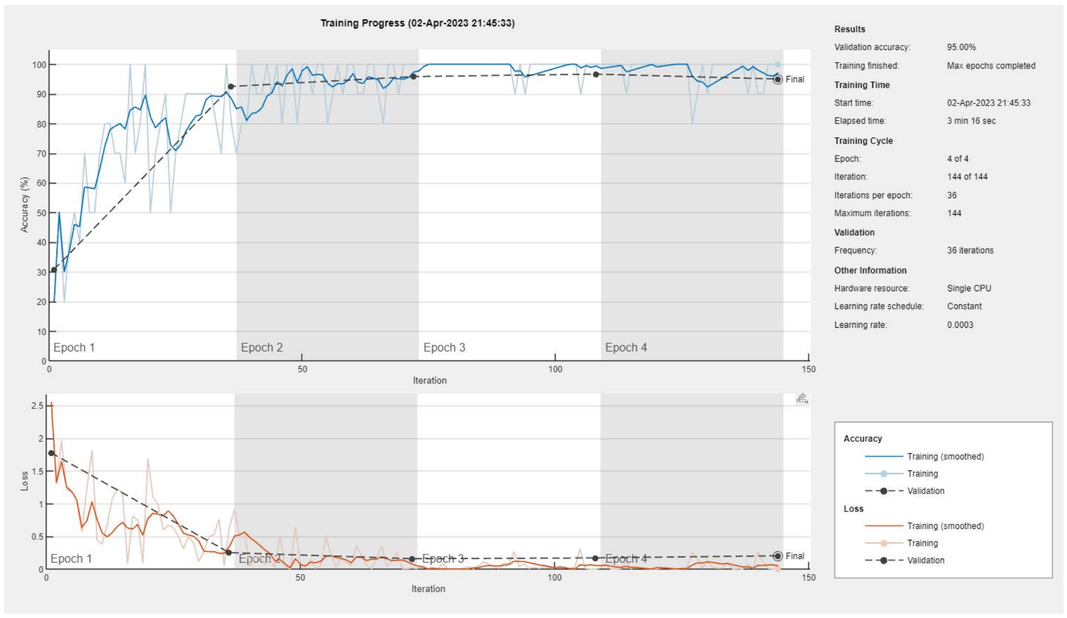


Figure 6.12: GradCAM for Normal Image versus GradCAM for Blurred Image

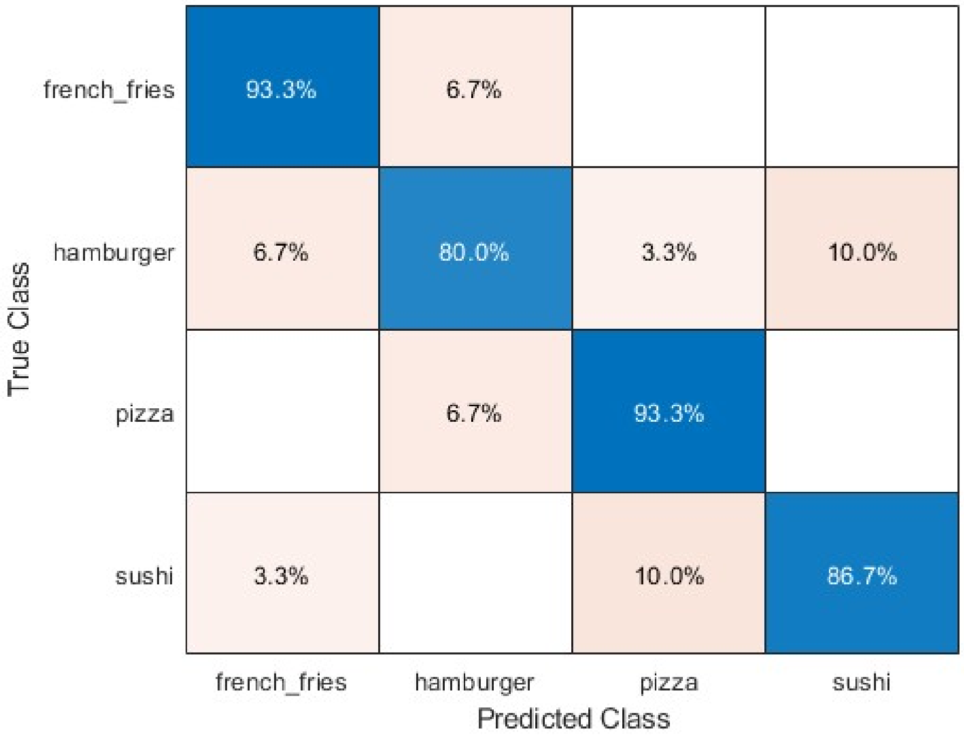
6.3 Augmented Trained Network for Blur

We want so see how we can improve the classification of blurred images. We will augment our training data by applying a slight blur to each of the images in our training data and add them to the training data. We now are training our network on 1200 images over 4 different classes. We then obtain the following training progress and confusion matrix:



Results
Validation accuracy: 95.00%
Training finished: Max epochs completed
Training Time
Start time: 02-Apr-2023 21:45:33
Elapsed time: 3 min 16 sec
Training Cycle
Epoch: 4 of 4
Iteration: 144 of 144
Iterations per epoch: 36
Maximum iterations: 144
Validation
Frequency: 36 iterations
Other Information
Hardware resource: Single CPU
Learning rate schedule: Constant
Learning rate: 0.0003

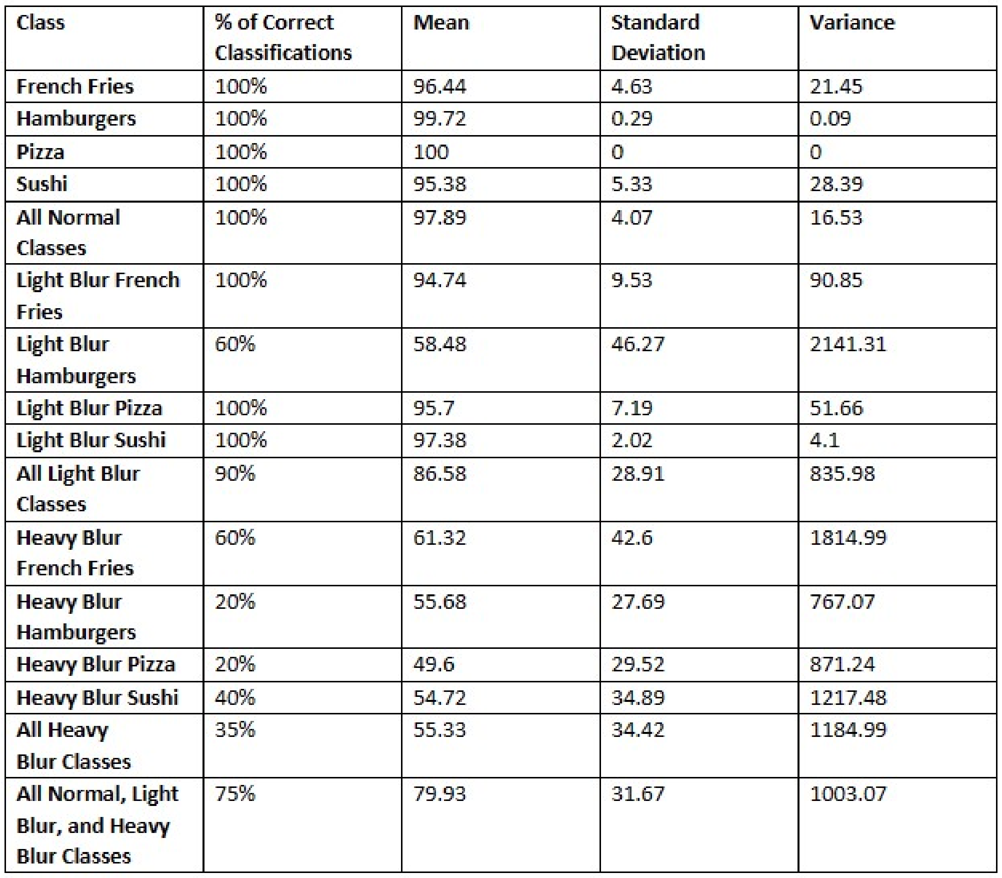
Figure 6.13: Training Progress for Augmented Network with Blur



	french_fries	hamburger	pizza	sushi
french_fries	93.3%	6.7%		
hamburger	6.7%	80.0%	3.3%	10.0%
pizza		6.7%	93.3%	
sushi	3.3%		10.0%	86.7%

Figure 6.14: Confusion matrix for Augmented Network with Blur

From the confusion matrix, we can clearly see improvement in the performance of our network. In particular, the sushi class has greatly improved. We now look at how the testing data performs:



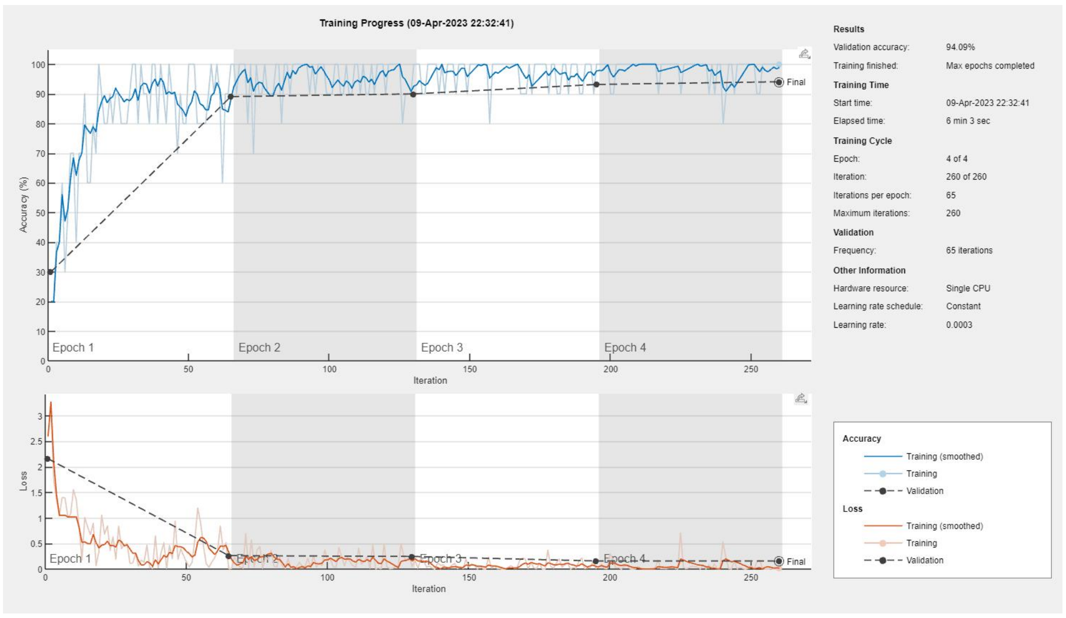
Class	% of Correct Classifications	Mean	Standard Deviation	Variance
French Fries	100%	96.44	4.63	21.45
Hamburgers	100%	99.72	0.29	0.09
Pizza	100%	100	0	0
Sushi	100%	95.38	5.33	28.39
All Normal Classes	100%	97.89	4.07	16.53
Light Blur French Fries	100%	94.74	9.53	90.85
Light Blur Hamburgers	60%	58.48	46.27	2141.31
Light Blur Pizza	100%	95.7	7.19	51.66
Light Blur Sushi	100%	97.38	2.02	4.1
All Light Blur Classes	90%	86.58	28.91	835.98
Heavy Blur French Fries	60%	61.32	42.6	1814.99
Heavy Blur Hamburgers	20%	55.68	27.69	767.07
Heavy Blur Pizza	20%	49.6	29.52	871.24
Heavy Blur Sushi	40%	54.72	34.89	1217.48
All Heavy Blur Classes	35%	55.33	34.42	1184.99
All Normal, Light Blur, and Heavy Blur Classes	75%	79.93	31.67	1003.07

Figure 6.15: Testing Statistics for Augmented Network for Blur

Comparing these results to the network trained solely on unaltered data, there is a significant improvement for both lightly blurred images and heavily blurred images. By augmenting our data, we were able to significantly improve performance.

6.4 Augmented Trained Network for Rotation

We will now look at improving the classification of rotated images. We will augment our data by applying different rotations on our training data and then adding those images to the training data. As in the blur case, our network will now be trained on 1200 images over four classes. We obtain the following training progress and confusion matrix:



Results	
Validation accuracy:	94.09%
Training finished:	Max epochs completed
Training Time	
Start time:	09-Apr-2023 22:32:41
Elapsed time:	6 min 3 sec
Training Cycle	
Epoch:	4 of 4
Iteration:	260 of 260
Iterations per epoch:	65
Maximum iterations:	260
Validation	
Frequency:	65 iterations
Other Information	
Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.0003

Figure 6.16: Training Progress for Augmented Network with Rotation

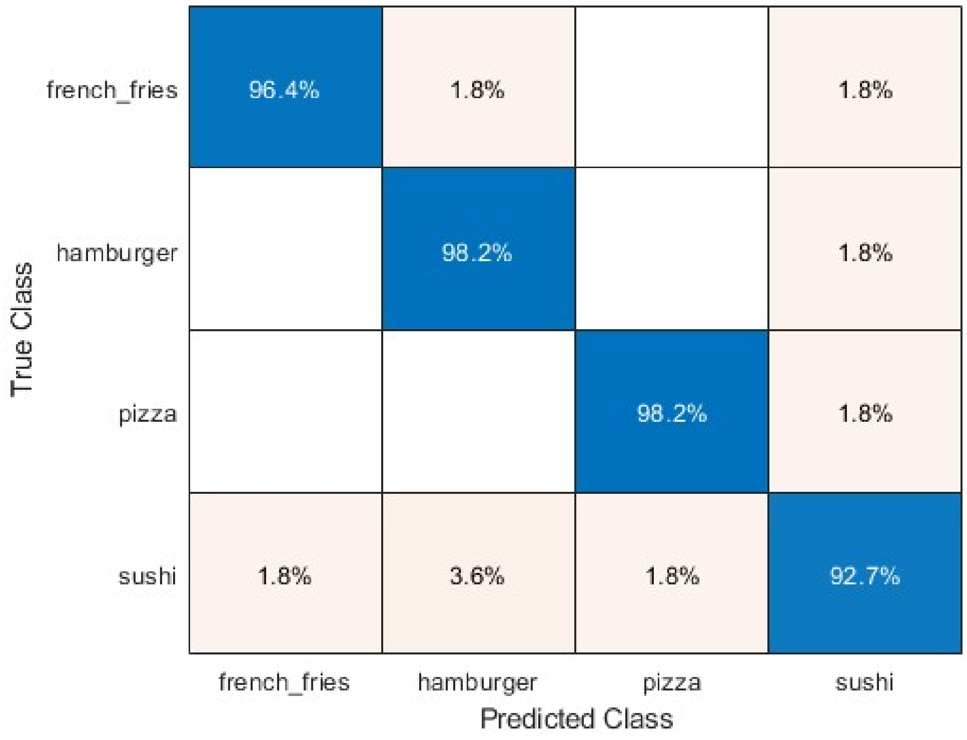
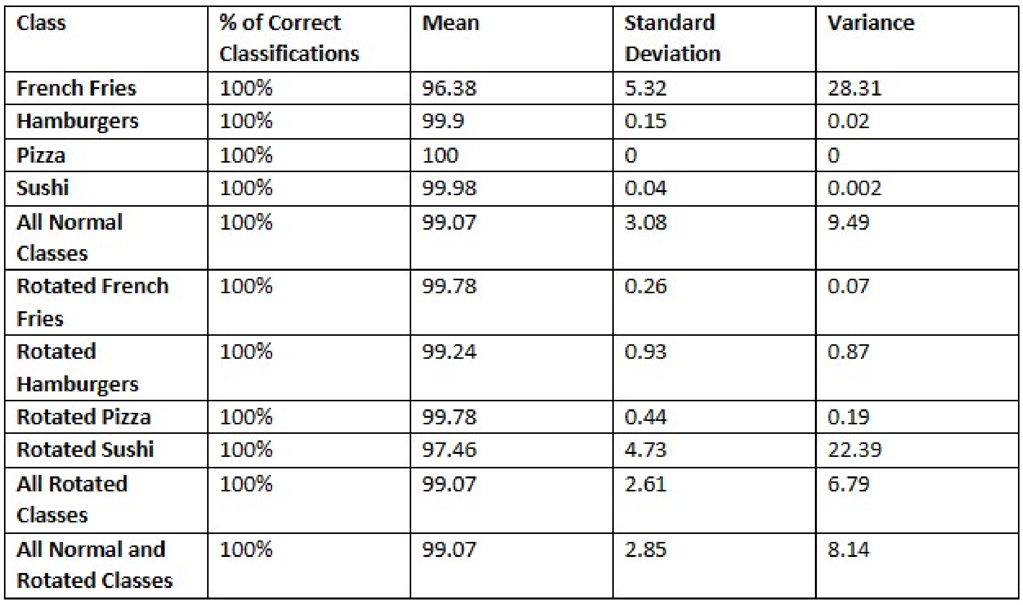


Figure 6.17: Confusion Matrix for Augmented Network with Rotation

We now run our independent testing set through the network;



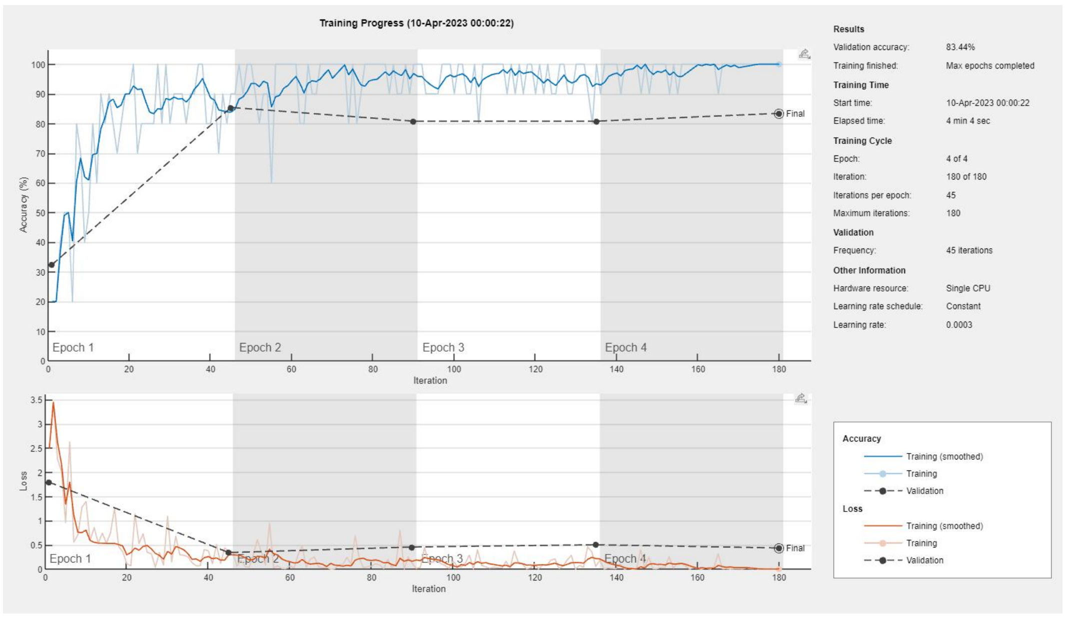
Class	% of Correct Classifications	Mean	Standard Deviation	Variance
French Fries	100%	96.38	5.32	28.31
Hamburgers	100%	99.9	0.15	0.02
Pizza	100%	100	0	0
Sushi	100%	99.98	0.04	0.002
All Normal Classes	100%	99.07	3.08	9.49
Rotated French Fries	100%	99.78	0.26	0.07
Rotated Hamburgers	100%	99.24	0.93	0.87
Rotated Pizza	100%	99.78	0.44	0.19
Rotated Sushi	100%	97.46	4.73	22.39
All Rotated Classes	100%	99.07	2.61	6.79
All Normal and Rotated Classes	100%	99.07	2.85	8.14

Figure 6.18: Testing Statistics for Augmented Network with Rotation

Once again, our network has improved. Even though our original network performed very well with rotated images, our augmented network performs even better. In our independent testing set, the rotated images performed just as well as the non-rotated images.

6.5 Augmented Trained Network for Translation

Finally, we want to see if we can improve the classification of translated images. This type of image performed poorly on our original network so improving this will be difficult. We augment our training data by translating our data and then adding it to the original training data. Our training data will once again consist of 1200 images. By training the network on this augmented data, we obtain the following training progress and confusion matrix:



Results	
Validation accuracy:	83.44%
Training finished:	Max epochs completed
Training Time	
Start time:	10-Apr-2023 00:00:22
Elapsed time:	4 min 4 sec
Training Cycle	
Epoch:	4 of 4
Iteration:	180 of 180
Iterations per epoch:	45
Maximum iterations:	180
Validation	
Frequency:	45 iterations
Other Information	
Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.0003

Figure 6.19: Training Progress for Augmented Network with Translation

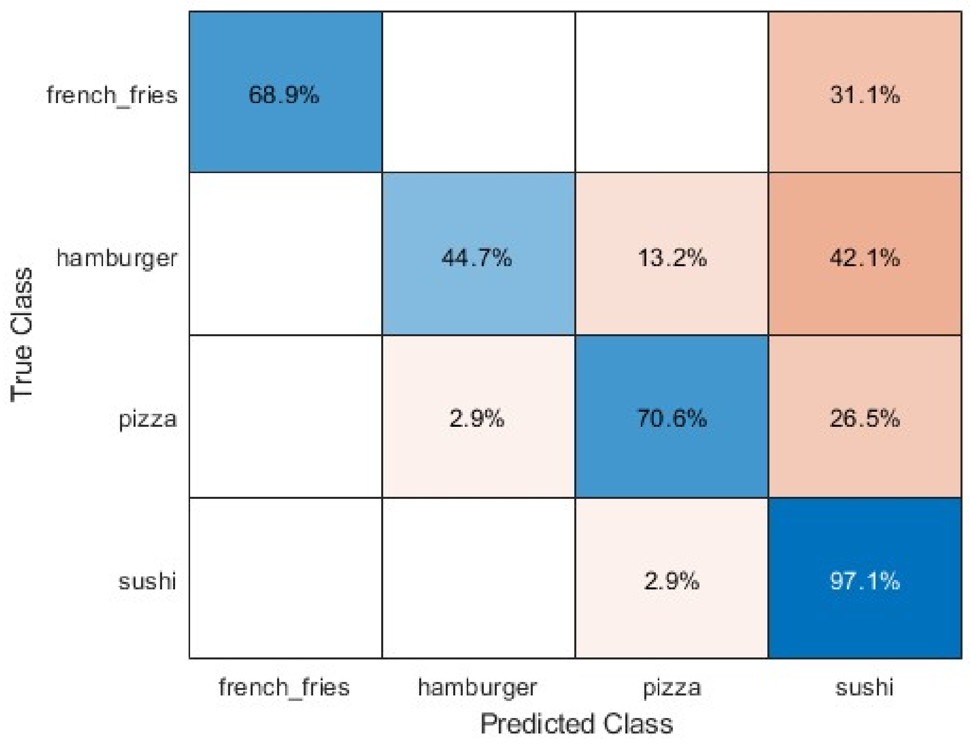
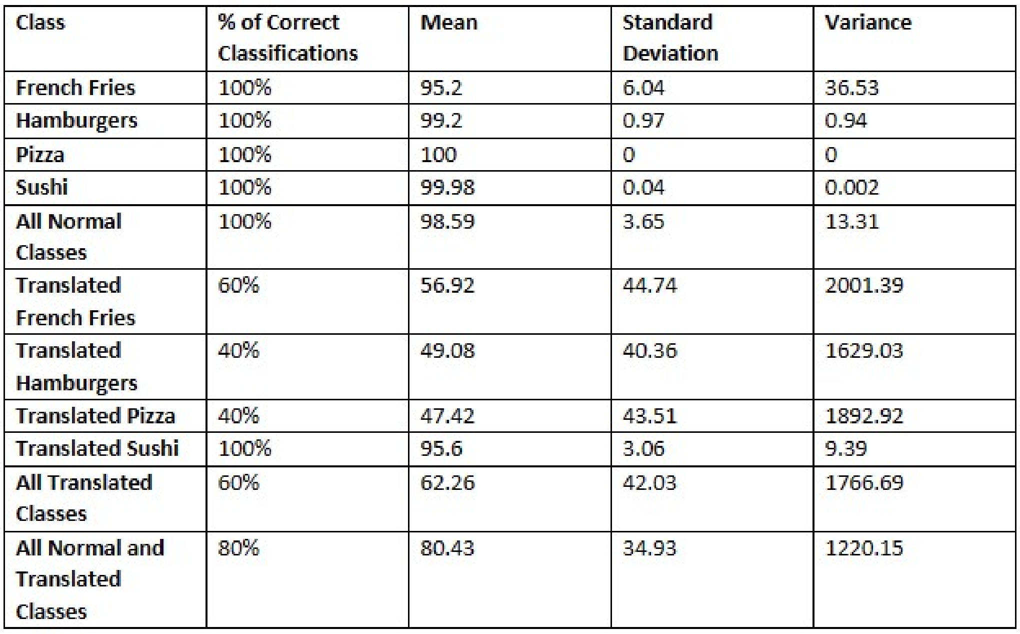


Figure 6.20: Confusion Matrix for Augmented Network with Translation

Interestingly, by augmented our training data with translated images, we have significantly improved the sushi class while significantly decreasing every other class. We now run our independent testing data on our augmented network:



Class	% of Correct Classifications	Mean	Standard Deviation	Variance
French Fries	100%	95.2	6.04	36.53
Hamburgers	100%	99.2	0.97	0.94
Pizza	100%	100	0	0
Sushi	100%	99.98	0.04	0.002
All Normal Classes	100%	98.59	3.65	13.31
Translated French Fries	60%	56.92	44.74	2001.39
Translated Hamburgers	40%	49.08	40.36	1629.03
Translated Pizza	40%	47.42	43.51	1892.92
Translated Sushi	100%	95.6	3.06	9.39
All Translated Classes	60%	62.26	42.03	1766.69
All Normal and Translated Classes	80%	80.43	34.93	1220.15

Figure 6.21: Testing Statistics for Augmented Network with Translation

Despite the alarming confusion matrix, the classification of translated images from our independent testing set has improved. Therefore, by augmenting the data with translated images, we can see a clear improvement in classification.

Chapter 7

Conclusion and Further Research

Empirically, augmenting our data improved performance in classification of blurred images, rotated images and translated images. Theoretically, however, we don't know if there is a limit to the improvement caused by augmentation. One potential area of further research would be trying to find a theoretical limit where no matter how much you augment your data, there will still be some error in classification. Another area of further research would be study training, in particular the backwards phase from a rigorous mathematical point of view.

Bibliography

[1] A. Boggess, F.J. Narcowich, *A First Course in Wavelets with Fourier Analysis*, second edition, Wiley, 2009.

[2] H.L. Royden, P.M. Fitzpatrick, *Real Analysis*, fourth edition, Pearson, 2010

[3] E. Stein, G. Weiss, *Introduction to Fourier Analysis on Euclidean Spaces*, first edition, Princeton, 1971.

[4] E. Bender, *Mathematical Methods in Artificial Intelligence*, first edition, IEEE, 1996

[5] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, MIT Press, 2016

[6] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, *Intriguing Properties of Neural Networks*, arXiv:1312.6199 (2013)

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, *Going Deeper with Convolutions*, arXiv:1409.4842 (2014)

[8] R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, D. Batra, *Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization*, arXiv:1610.02391 (2016)

[9] M. Lin, Q. Chen, S. Yan, *Network in Network*, arXiv:1312.4400 (2014)

[10] N. Cui, *Applying Gradient Descent in Convolutional Neural Networks*, Journal of Physics: Conference Series, (2018)

Acknowledgements

There are a few people that I could not have made this thesis without. First, my advisor Gabriel Picioroaga, who provided immeasurable support and guidance during this process. Second, to my beautiful fiancée Faith Guzzo, who tirelessly supported me through the late nights and weekends working and whose love could light up the night sky. Third, to my family in Pennsylvania, whose love and support for me in this journey kept me going. Fourth, to my dog Zeus, who provided the best stress relief on the planet. Fifth, to my classmate and good friend Drake Hill, who was the best desk buddy one could ask for. Finally, to sunflower seeds, whose salty flavor helped propel me to write this thesis. I could not have done this without all of you. Thank you.