Neuro. Homework 6.

April 25, 2016

1 Neural Network Hw6 Write Up and Code

Hi! In this document you'll find all of my code and a paper I wrote on a generalization of neural networks called GANNs. I prove that GANNs are a strict superset of all neural networks and infact I provide a general backpropagation algorithm whose derivation is attached as the next document (a supplement to the statement I make in the paper).

I hope that suffices as an answer to the first question. Of the assignment.

In [2]:
```
%matplotlib inline
import scipy.io
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
from IPython import display
import random

%matplotlib inline
```

In [3]:
```
# LOAD TRAINING DATA
train_mat = scipy.io.loadmat('train.mat')
train_images_raw = train_mat['train_images'].T
train_images_corrected = []
for i in range(len(train_images_raw)):
    train_images_corrected.append(train_images_raw[i].T.ravel())
train_images_corrected = np.array(train_images_corrected)
train_labels_corrected = train_mat['train_labels'].ravel()

#LOAD TESTING DATA

test_mat = scipy.io.loadmat('test.mat')
test_images_raw =test_mat['test_images']
test_images_corrected = []
for i in range(len(test_images_raw)):
    test_images_corrected.append(test_images_raw[i].ravel())
test_images_corrected = np.array(test_images_corrected)
```

In [4]:
```
# Preprocess the data
def normalize(X):
    Xprime = (X /255.0)

    return Xprime

def sep_categories(Y, rangeY):
      return np.array([np.array([(i == y) for i in range(rangeY)])*1 for y in Y])
```

```
X_tr = normalize(train_images_corrected)
Y_tr = sep_categories(train_labels_corrected, 10)
X_te = normalize(test_images_corrected)

#plt.imshow(X_tr[14515].reshape(28,28), interpolation='nearest', cmap=plt.cm.ocean, extent=(0.5,
#plt.colorbar()
#plt.show()
```

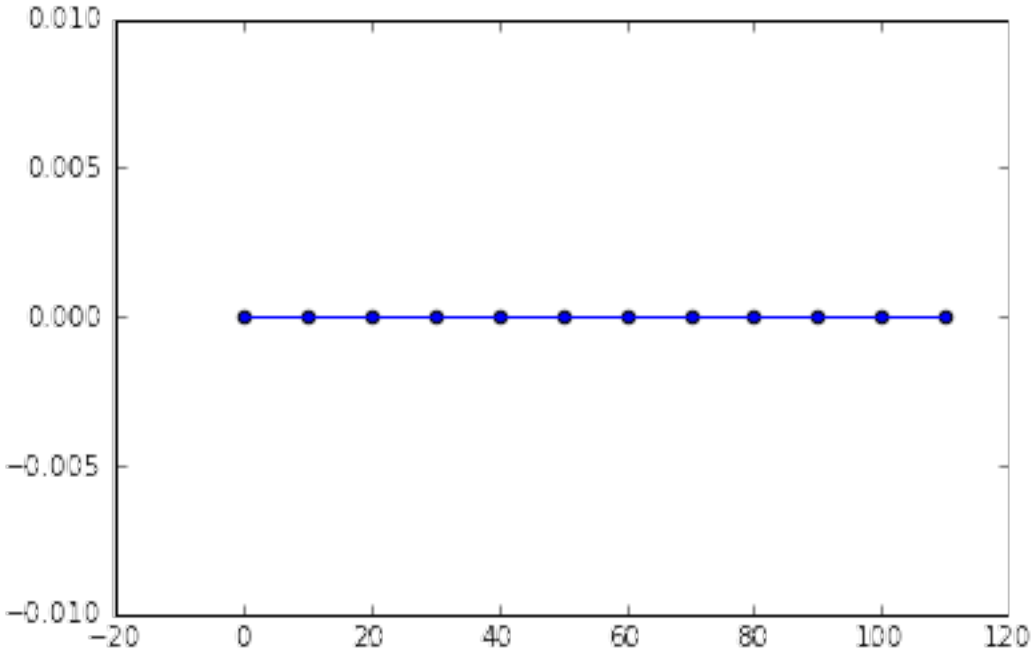
In [5]:
```
# Data separation
shuffle = np.random.permutation(np.arange(len(X_tr)))
X_tr, Y_tr = X_tr[shuffle], Y_tr[shuffle]

validation_count = 10000
X_va, Y_va = X_tr[0:validation_count], Y_tr[0:validation_count]
X_tr, Y_tr = X_tr[validation_count:], Y_tr[validation_count:]
```

In [6]:
```
class sigmoid:
    def __init__(self, func, derivative):
        self.func = func
        self.prime = derivative
    def __call__(self, x):
        return self.func(x)
logistic = sigmoid(scipy.special.expit, lambda x: scipy.special.expit(x)*(1 -scipy.special.expit

def hat(x):
    return np.append(x, [1])

def scratch(mat):
    return mat[:-1, :] #Todo improve speed to O(1)


# Neural network
class ann:
    def __init__(self, neuro_c, sigma=logistic):
        self.L = len(neuro_c) - 1
        self.W = [None] * (len(neuro_c) - 1)
        self.net = [None] * (len(neuro_c) -1)
        self.O = [None] * len(neuro_c)

        self.sigma = sigma


        for i in range(len(neuro_c)-1):
            self.W[i] = np.random.rand(neuro_c[i]+1, neuro_c[i+1]) * 2 -1 # 1 for bias


    """Performs the feedforward portion of the algorithm. Returns the output."""
    def feed_forward(self, x):
        self.O[0] = x
        for i, w in enumerate(self.W):
            #bias
            Ihat = hat(self.O[i])
            self.net[i] = w.T.dot(Ihat)
```

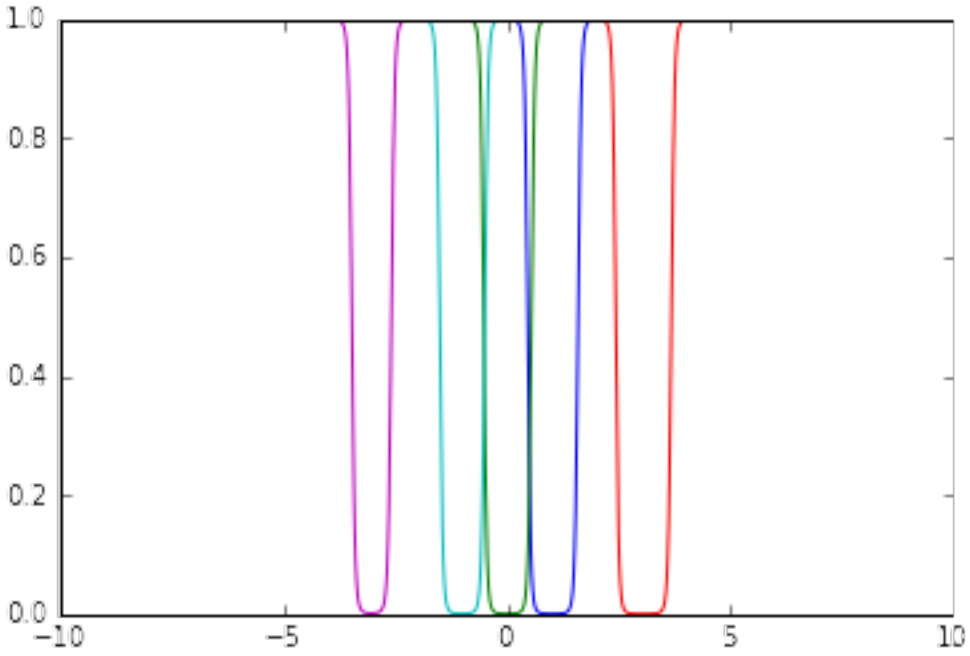
```
            self.O[i+1] = np.apply_along_axis(self.sigma,0, self.net[i])

        return self.O[self.L]

    """Performs error backpropagation for calculation of gradW. Returns a list of matrices."""
    def backpropagate(self, y, loss):
        delta = [None] * (self.L)

        if loss == "L2":
            delta[self.L - 1] = (self.O[self.L] - y) \
                * np.apply_along_axis(self.sigma.prime,0,self.net[self.L-1])
        elif loss == "cross_entropy":
            delta[self.L - 1] = (self.O[self.L] - y)
        else:
            raise Exception("wtf i dont kno that loss function")


        for l in reversed(range(self.L -1)):
            delta[l] = (scratch(self.W[l+1]).dot(delta[l+1]) \
                       * np.apply_along_axis(self.sigma.prime,0,self.net[l]))


        gradW = list(map(lambda l: np.outer(hat(self.O[l]), delta[l]), range(self.L)))

        return gradW

    """Calculates the gradient of the neuralnetwork w.r.t. each weight matrix."""
    def gradient(self, dp, loss="L2"):
        x,y = dp
        self.feed_forward(x)
        return self.backpropagate(y, loss)

    """Performs minibatch stochastic gradient descent."""
    def msgd(self, dataset, loss="L2", batchsize=10, lr=0.5, decay=0):
        while True:
            #dataset = random.sample(dataset, len(dataset))
            #for each minibatch
            for q in range(int(len(dataset)/batchsize)):
                batch = dataset[q*batchsize:(q+1)*batchsize]
                # get gradient
                gradW = list(map(lambda dp: self.gradient(dp, loss), batch))
                # apply weight update rule (w/O decay)
                # remember gradW is a list of list of Ws for each weight matrix
                totgradW = [0] * (self.L)
                for i in range(len(totgradW)):
                    for j in range(len(gradW)):

                        totgradW[i] += gradW[j][i]

                for i,w in enumerate(self.W):
                    self.W[i] =(1 - decay/float(len(dataset)))*w - lr*totgradW[i]


            yield self.error(dataset)
```

```
        def error(self, dataset):
            err1 = 0
            err2 = 0
            for dp in dataset:
                out = self.feed_forward(dp[0])
                if np.argmax(out) != np.argmax(dp[1]):
                    err1 += 1
                err2 += np.linalg.norm(out - dp[1])

            return err1/len(dataset)

        def __call__(self, x):
            return self.feed_forward([x])

    """Plots a generator interactively"""
    def interactive_plot(generator, epochs, freq=10, comparison=None):
        ep = []
        er = []
        cmp= []
        for epoch in range(epochs):
            try:
                err = next(generator)
                if(epoch % freq == 0):
                    ep.append(epoch)
                    er.append(err)
                    plt.scatter(epoch,err)
                    if comparison is not None:
                        cmp_n =  next(comparison)
                        plt.scatter(epoch, cmp_n, color='red')
                        cmp.append(cmp_n)
                    display.display(plt.gcf())
                    display.clear_output(wait=True)
            except KeyboardInterrupt:
                break
        plt.plot(ep,er)
        if comparison is not None:
            plt.plot(ep, cmp, color='red')
        return ep,er

    # MNIST extravaganza
    """Gets error on the validation set"""
    def validation_error(net):
        while True:
            yield net.error(list(zip(X_va,Y_va)))
```

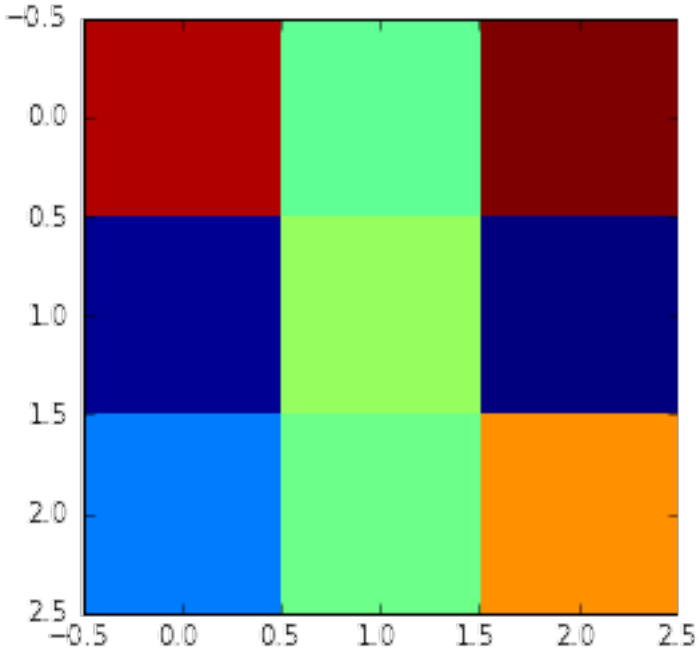
In [418]:
```
tester = ann([2,3,4,1])

gradW = [tester.gradient(([0,1],1)), tester.gradient(([1,0],1))]
xordata = [([1,0], [1]), ([0,0], [0]), ([0,1],[1]), ([1,1],[0])]

optimizer  = tester.msgd(xordata, loss="cross_entropy", lr=0.6, batchsize=2)
pltdat = interactive_plot(optimizer, 3000, freq=10)
plt.show()
```




```
In [403]: x = np.linspace(-10, 10, 1000)
          y = np.array([tester.feed_forward([xi, 1]) for xi in x])
          plt.plot(x, y)
          y = np.array([tester.feed_forward([xi, 0]) for xi in x])
          plt.plot(x, y)
          y = np.array([tester.feed_forward([xi, 3]) for xi in x])
          plt.plot(x, y)
          y = np.array([tester.feed_forward([xi, -1]) for xi in x])
          plt.plot(x, y)
          y = np.array([tester.feed_forward([xi, -3]) for xi in x])
          plt.plot(x, y)
          plt.show()

          plt.imshow(tester.W[0], interpolation='nearest')
```

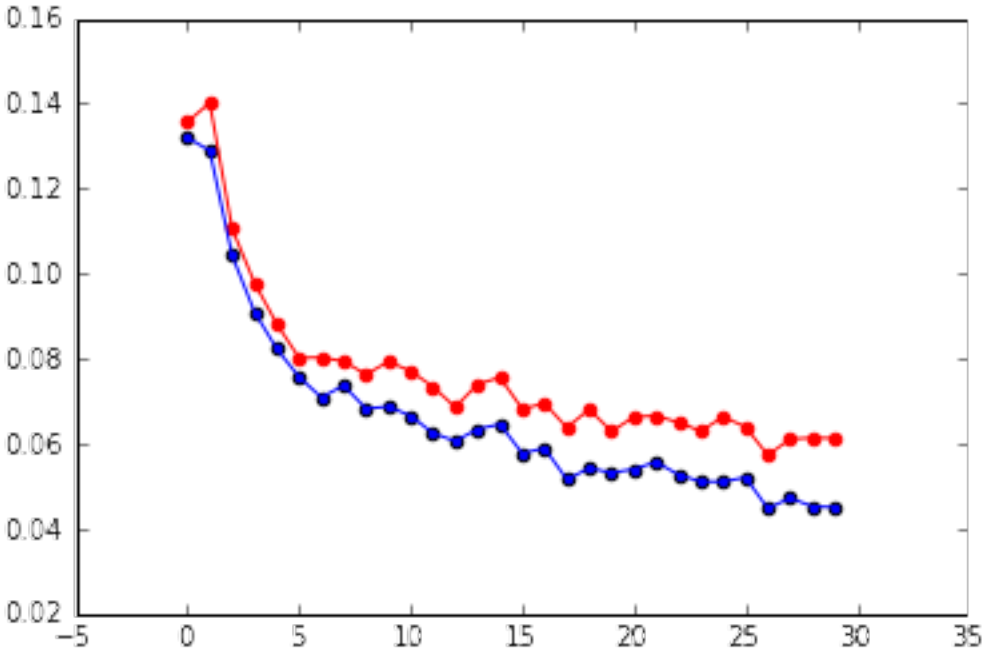


Out [403]: <matplotlib.image.AxesImage at 0x20a98edf780>



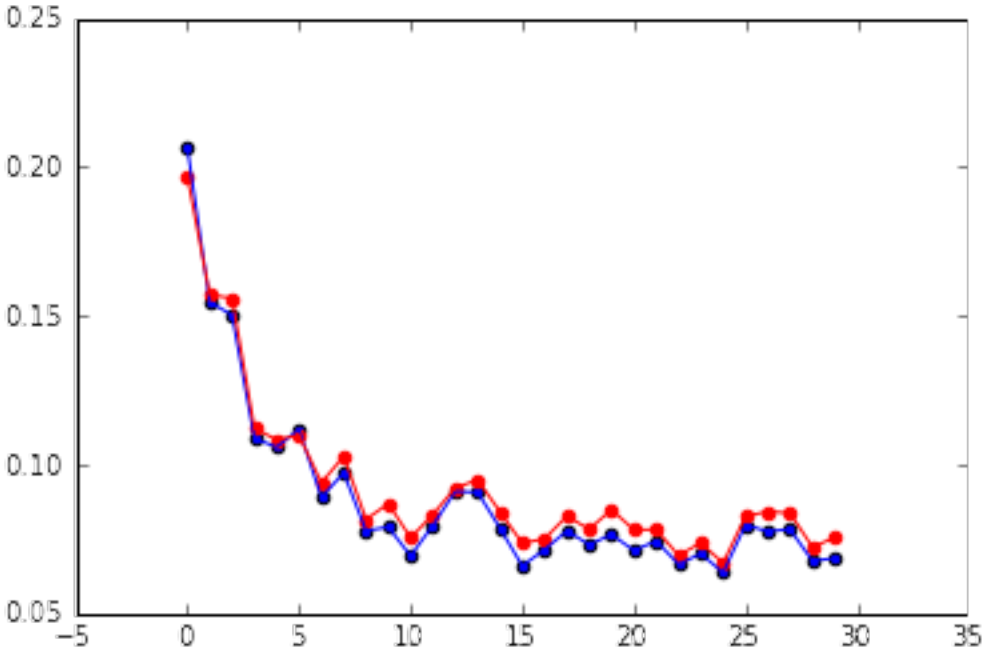
In [421]: mnistNet = ann([784,100,10])

```
optimizer  = mnistNet.msgd(list(zip(X_tr,Y_tr)), loss="cross_entropy", lr=0.5, batchsize=10)
pltdat = interactive_plot(optimizer, 30, freq=1, comparison=validation_error(mnistNet))
```

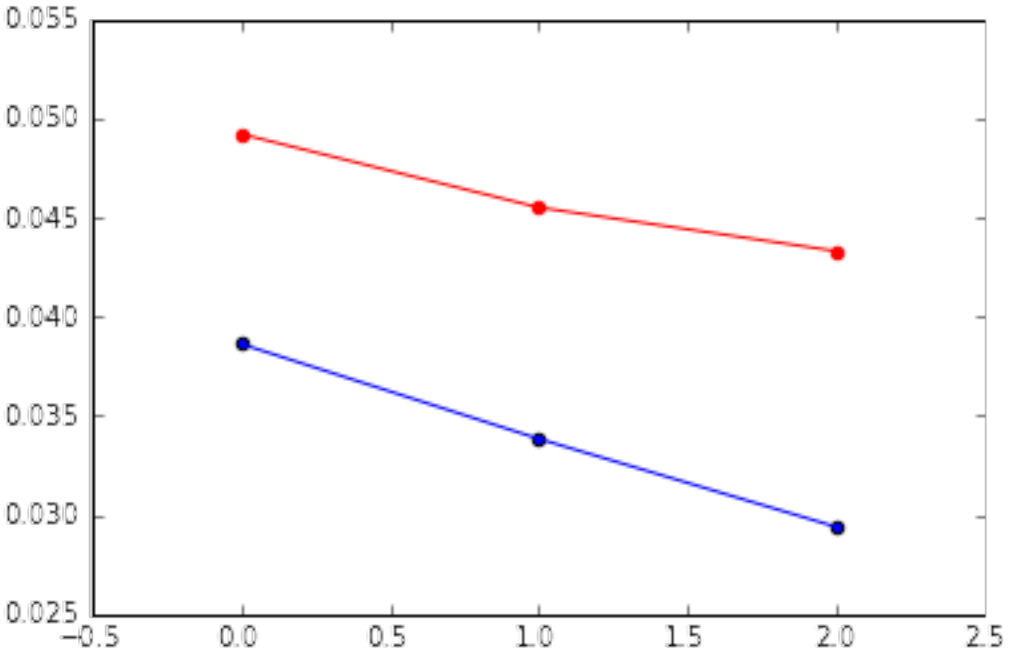


In [12]:
```
mnistNet = ann([784,200,10])
```

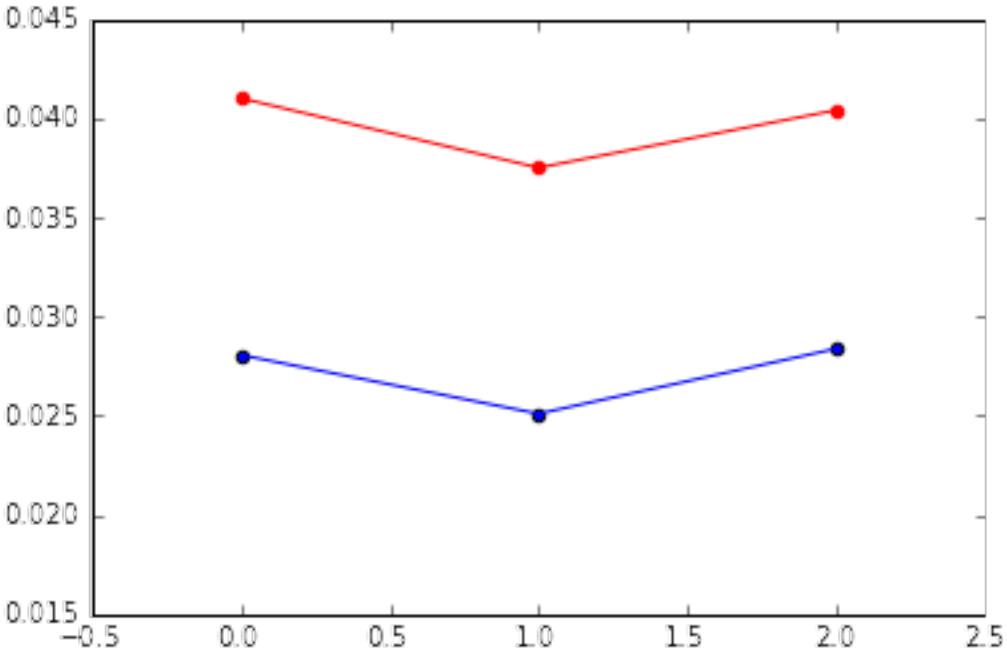
```
optimizer  = mnistNet.msgd(list(zip(X_tr,Y_tr)), loss="cross_entropy", lr=0.5, batchsize=10, de
pltdat = interactive_plot(optimizer, 30, freq=1, comparison=validation_error(mnistNet))
```



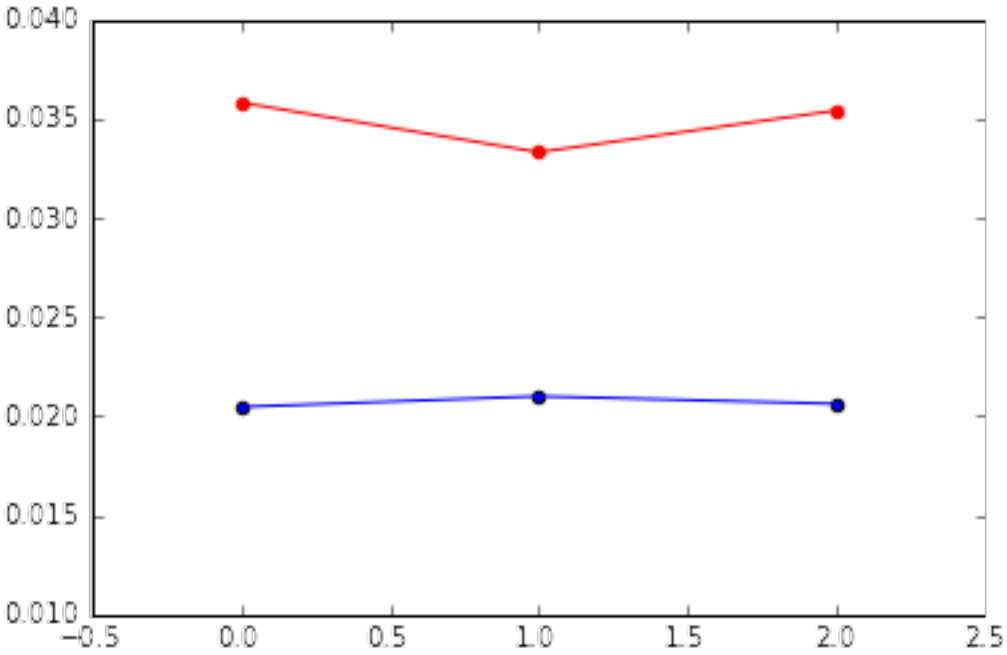
In [13]:
```
# drop out to a lower learning rate; let's be careful!
careful_optimizer  = mnistNet.msgd(list(zip(X_tr,Y_tr)), loss="cross_entropy", lr=0.25, batchs
pltdat = interactive_plot(careful_optimizer, 3, freq=1, comparison=validation_error(mnistNet))
```



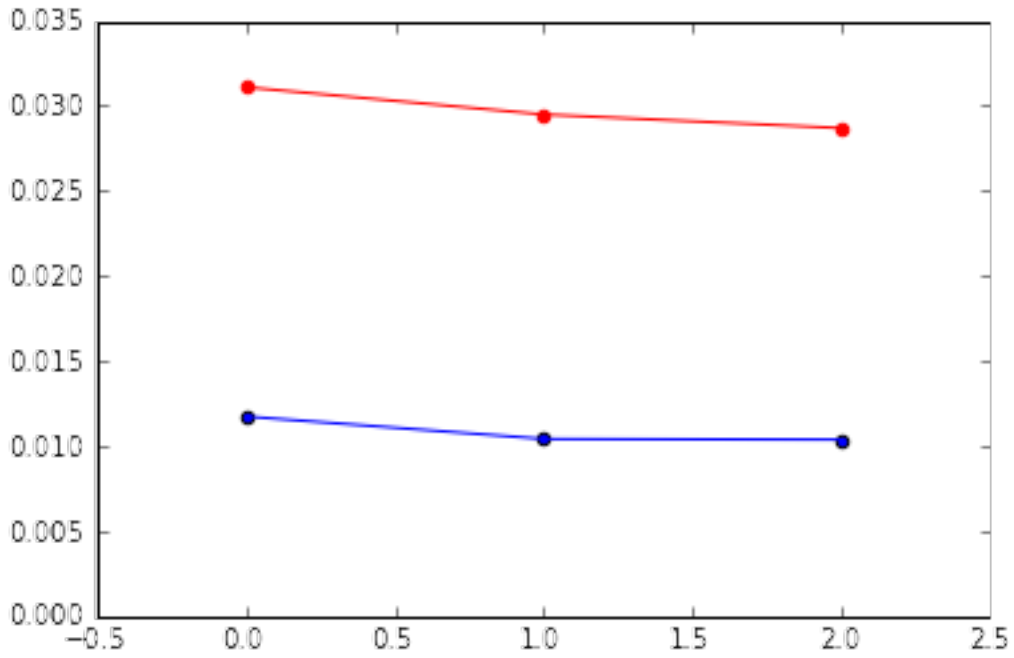
In [14]:
```
careful_optimizer  = mnistNet.msgd(list(zip(X_tr,Y_tr)), loss="cross_entropy", lr=0.25, batchs
pltdat = interactive_plot(careful_optimizer, 3, freq=1, comparison=validation_error(mnistNet))
```



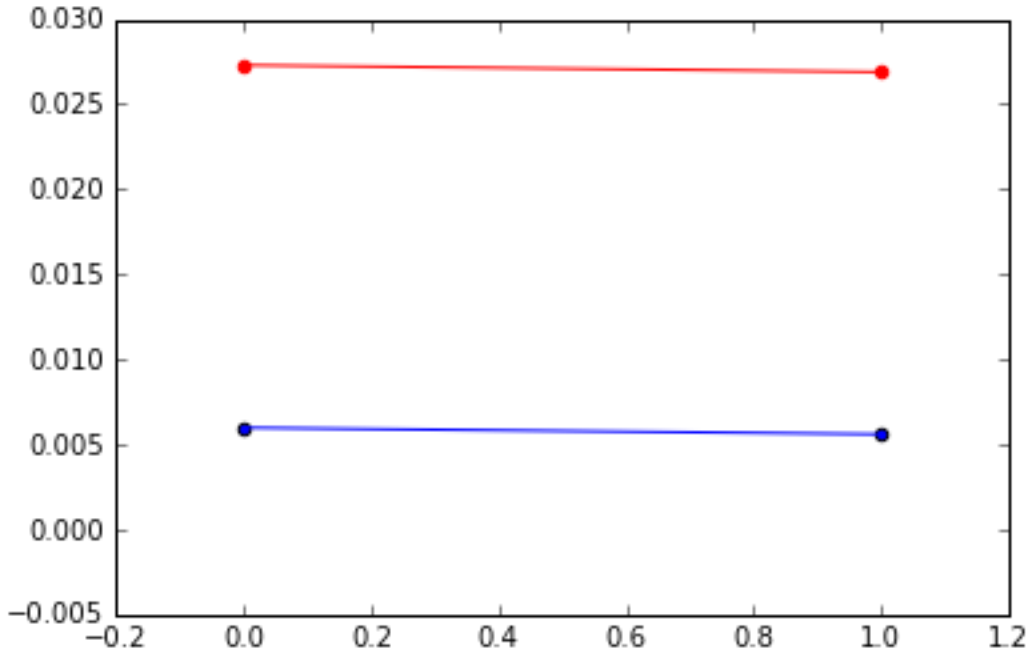
In [15]:
```
careful_optimizer  = mnistNet.msgd(list(zip(X_tr,Y_tr)), loss="cross_entropy", lr=0.20, batchs
pltdat = interactive_plot(careful_optimizer, 3, freq=1, comparison=validation_error(mnistNet))
```



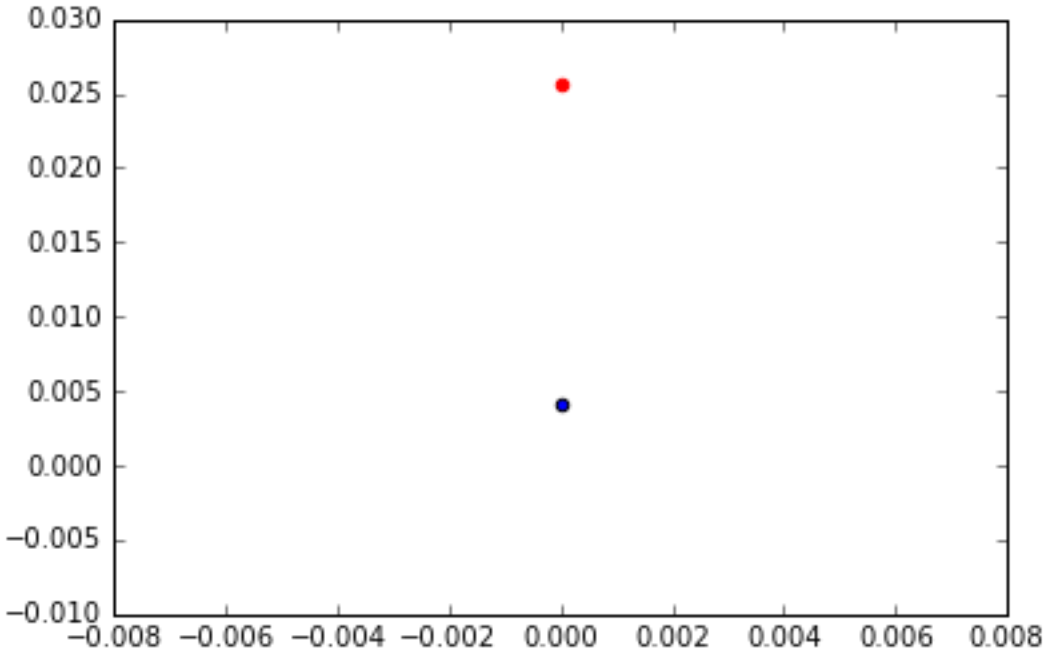
In [16]:
```
careful_optimizer  = mnistNet.msgd(list(zip(X_tr,Y_tr)), loss="cross_entropy", lr=0.15, batchs
pltdat = interactive_plot(careful_optimizer, 3, freq=1, comparison=validation_error(mnistNet))
```



In [18]:
```
careful_optimizer  = mnistNet.msgd(list(zip(X_tr,Y_tr)), loss="cross_entropy", lr=0.10, batchs
pltdat = interactive_plot(careful_optimizer, 3, freq=1, comparison=validation_error(mnistNet))
```



In [19]:
```
careful_optimizer  = mnistNet.msgd(list(zip(X_tr,Y_tr)), loss="cross_entropy", lr=0.07, batchs
pltdat = interactive_plot(careful_optimizer, 3, freq=1, comparison=validation_error(mnistNet))
```




```
In [20]: outp = list(map(lambda dp:  mnistNet.feed_forward(dp), X_te))

In [22]: outp = list(map(lambda pred: np.argmax(pred), outp))

In [25]: kaggle = pd.DataFrame(outp, columns=["Category"])

In [27]: kaggle.to_csv('kaggle.csv')
```

Dimensionality Reduction Using Generalized Artificial Neural Networks

William Guss WGUSS@BERKELEY.EDU
Machine Learning at Berkeley, 246 Cory Hall, Berkeley, CA 94720 USA

Abstract

In this paper, we generalize ANNs to infinite dimensional Banach spaces by developing a practical analog to the feedforward propagation algorithm. Using this new class of algorithms, GANN, we prove a new universal approximation theorem for bounded linear operators and show that representation of weights by samples from a multivariate weight polynomial can drastically reduce the dimensionality of a learning problem. Lastly, we give a practical implementation of the error back-propagation algorithm in this space for the classification of continuous data.

1. Introduction

Although neural networks have proven an extremely effective mechanism of machine learning (Burch, 2001), theoretically they remain a black-box model. In answer to this problem (Neal, 2012) examined the notion of infinite hidden nodes with a network proving that such a construction becomes a Gaussian kernel. Then, (Roux & Bengio, 2007) described a model for affine neural networks with continuous hidden layers in alignment with (Neal, 2012). These authors showed effectively the viability of a "continuous" neural network, but left many similar constructions unexplored.

It is the subject of this paper to generalize the construction of a feed-forward ANN which maps uncountably infinite vector spaces, and then to demonstrate the practical implementations of algorithms such as error backpropagation in this generalized form.

2. Functional Neural Networks

In the standard feed-forward case proposed in (McCulloch & Pitts, 1943), if on the lth layer, $Z^{(l)} = 1, \ldots, n$ is the index set of neurons, β is the bias, g is the sigmoid activation function, and σ_j is the output of the jth neuron, the following recurrence relation is natural.

Definition 1. *We say $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^m$ is a feed-forward neural network if for an input vector $\boldsymbol{x}$,*

$$\begin{aligned} \mathcal{N} : \sigma_j^{(l+1)} &= g\left(\sum_{i \in Z^{(l)}} w_{ij}^{(l)} \sigma_i^{(l)} + \beta^{(l)}\right) \\ \sigma_j^{(1)} &= g\left(\sum_{i \in Z^{(0)}} w_{ij}^{(0)} x_i + \beta^{(0)}\right), \end{aligned} \tag{1}$$

where $1 \leq l \leq L-1$. Furthermore we say $\{\mathcal{N}\}$ is the set of all neural networks.

Suppose that we wish to map one functional space to another with a neural network. Consider the standard model of an ANN as the number of neural nodes for every layer becomes uncountable. The index for each node then becomes real-valued, along with the weight and input vectors.

Let us denote $\xi : X \subset \mathbb{R} \to \mathbb{R}$ as some arbitrary input function for the neural network. Likewise consider a real-valued weight function, $w^{(l)} : \mathbb{R}^2 \to \mathbb{R}$, for a layer l which is composed of two indexing variables $i, j \in \mathbb{R}$. Finally as the number of neural nodes becomes uncountable we define a real-valued bound for any given layer $R^{(l)} \supset Z^{(l)}$. As the indices become real valued, examination of the weighted sum seen in Definition 1 leads us to the following abuse of notation which is not too dissimilar from the derivation of the Laplace Transform from the power series.

$$\begin{aligned} \sigma^{(1)}(j) &= \lim_{\Delta i \to 0} g\left(\sum_{i \in R^{(0)}} \xi(i) w^{(0)}(i,j)\ \Delta i + \beta\right) \\ &= g\left(\int_{R^{(0)}} \xi(i) w^{(0)}(i,j)\, di\right) \end{aligned} \tag{2}$$

The β bias term is omitted as it can surely be formed as an artifact of integration of the weight function. Repeating the process inductively for all layers in a neural network, leads to a recurrence relation for this construction.

Proceedings of the 33^{rd} International Conference on Machine Learning, New York, NY, USA, 2016. JMLR: W&CP volume 48.

Definition 2. *We call $\mathcal{F} : L^1(X) \to L^1(Y)$ a functional neural network if,*

$$\begin{aligned} \mathcal{F} : \sigma^{(l+1)}(j) &= g\left(\int_{R^{(l)}} \sigma^{(l)}(i) w^{(l)}(i,j)\, di\right) \\ \sigma^{(0)}(j) &= \xi(j). \end{aligned} \tag{3}$$

Furthermore let $\{\mathcal{F}\}$ denote the set of all functional neural networks.

To demonstrate the accuracy of this generalization, let us propose the following theorem which is near that of (Roux & Bengio, 2007).

Theorem 3. *Suppose $\mathcal{F}$ is a functional neural network with a set of piecewise constant weight functions $W = \left\{w^{(0)}, w^{(1)}, \ldots, w^{(L-1)}\right\}$ each with constituent pieces of length one. Then given the input function $\xi(i) = x_n, n = \max\{m \in \mathbb{Z} \mid m \leq i\}$ for some input vector $\boldsymbol{x}$, $\mathcal{F}$ is discretized; that is assuming $i, j \in \mathbb{R}$ and $Z^{(l)} = R^{(l)} \cap \mathbb{Z}$ then for $0 \leq l \leq L-1$ we have that*

$$\mathcal{F}(\xi) = \mathcal{N}(\boldsymbol{x}), \tag{4}$$

so $\{\mathcal{F}\} \supset \{\mathcal{N}\}$

Proof. Let $P(l)$ be the proposition that $\sigma^{(l+1)}(j)$ becomes discretized when $w^{(l)}(i,j)$ and $\sigma^{(l)}(i,j)$ are piecewise constant with constituent functions of length one. Moreover let $w_{ij}^{(l)}$ denote the value of $w^{(l)}(i,j)$ for $(i,j) = \max\left\{(x,y) \in \mathbb{Z}^2 \mid x \leq i \wedge y \leq j\right\}$. Then by induction we show $P(l), 0 \leq l \leq L-1$.

Basis Step. Recall that $\sigma^{(0)}(j) = \xi(j)$. Then one would suppose

$$\sigma^{(1)}(j) = g\left(\int_{R^{(0)}} \xi(i) w^{(0)}(i,j)\, di+\right) \tag{5}$$

but because the weight function and the input function are piecewise constant and not guaranteed to be continuous for $R^{(0)}$, we must take the improper integral along each constituent piece of length one. Supposing that each summation in the following is taken over $k \in Z^{(0)}$,

$$\begin{aligned} \sigma^{(1)}(j) &= g\left(\sum_k \lim_{b \to k} \int_{k-1}^{b} \xi(i) w^{(0)}(i,j)\, di+\right) \\ &= g\left(\sum_k w_{kj}^{(0)} \lim_{b \to k} \int_{k-1}^{b} \xi(i)\, di+\right) \\ &= g\left(\sum_k w_{kj}^{(0)} x_b+\right) \end{aligned} \tag{6}$$

Inductive Step. Now assume that for some l we have that

$$\sigma^{(l+1)}(j) = g\left(\sum_{i \in Z^{(l)}} w_{ij}^{(l)} \sigma_i^{(l)}+\right) \tag{7}$$

We now show that by the inductive hypothesis if $P(0) \wedge P(l) \to P(l+1)$, then $P(k)\ \forall k$. Consider the next neural layer defined as

$$\sigma^{(l+2)}(j) = g\left(\int_{R^{(l+1)}} \sigma^{(l+1)}(i) w(i,j)\, di+\right) \tag{8}$$

Then because $w(i,j)$ and $\sigma^{(l+1)}$ are piecewise constant by definition and not necessarily continuous for $R^{(l+1)}$ we must again take the improper Riemann integral over the constituent pieces. Consider $k \in Z^{(l+1)}$

$$\begin{aligned} \sigma^{(l+2)}(j) &= g\left(\sum_k \lim_{b \to k} \int_{k-1}^{b} \sigma^{(l+1)}(i) w(i,j)\, di+\right) \\ &= g\left(\sum_k w_{kj}^{(l+1)} \sigma_k^{(l+1)} \lim_{b \to k} \int_{k-1}^{b} di+\right) \\ &= g\left(\sum_k w_{kj}^{(l+1)} \sigma_k^{(l+1)}+\right) \end{aligned} \tag{9}$$

Therefore by the inductive hypothesis the proof follows and $\mathcal{F}(\xi) = \mathcal{N}(\boldsymbol{x})$ for piecewise constant input and weight functions.

□

From the logic of the preceding proof we can establish that the input function need only be properly Lebesgue integrable over $R^{(0)}$. Moreover, we come to an extremely important intuition; the weight matrix for a given layer l can be thought of as a piecewise constant weight surface, and the linear combination of weights can be thought of as a piecewise integral transformation along a given j on the weight surface. However, functional neural networks allow for infinite weight surfaces and therefore can represent the entire class of integral transforms. With this in mind, it is now possible to consider functional neural networks as universal approximators.

3. Universal Approximation of Bounded Linear Operators

In the case of discretized neural networks, George Cybenko and Kolmogorov have shown that with sufficient weights and connections, a feed-forward neural network is a universal approximator of arbitrary $f : \mathbb{R}^n \to \mathbb{R}^m$ (Cybenko, 1989); that is, constructs of the form $\mathcal{N}(\boldsymbol{x})$ are dense in $C(I^n, \mathbb{R}^m)$ where I^n is the unit hypercube $[0,1]^n$. Cybenko proved this remarkable result by utilizing the Riesz Representation Theorem for Hilbert spaces and the Hahn-Banach theorem. He showed by contradiction that there exists no bounded linear functional $h(x)$ in the form of $\mathcal{N}(\boldsymbol{x})$ such that $\int_{I_n} h(x)\, d\mu(x) = 0$.

Although this is result legitimizes neural networks, it is rather vacuous since it does not actually impart constraints on the type of networks which might approximate these functions. Fortunately using the intuitions presented in Theorem 1, it would be advantageous to examine the generalization of Cybenko's theorem to the larger class of functional neural networks. However, there is no clear way with which to do this; discretized neural networks map vector spaces and therefore approximate continuous functions, whereas functional neural networks are defined as arbitrary mappings between Hilbert spaces (more specifically the set of L^2 integrable functions). Moreover letting n approach infinity in Cybenko's proof fails to hold in that there is not an obvious topology for $C(C(\mathbb{R}), C(\mathbb{R}))$ which satisfies it. Therefore we must develop an approximation theorem for $\mathcal{F} : C(X) \to C(Y)$ over the set of linear operators.

First, however, let us develop the notion of $\mathcal{F}$ as a universal approximator of arbitrary functions. By Theorem 1, we have that $\{\mathcal{F}\} \supset \{\mathcal{N}\}$,and in that sense for piecewise constant $w(i, j)$ and $\xi(i)$ functional neural networks approximate any arbitrary mapping from $\mathbb{R}^n \to \mathbb{R}$ where $n = |Z^{(0)}|, m = |Z^{(L-1)}|$ by Cybenko's theorem. However, when considering the fully continuous case the following corollary arises from the Stone-Weierstrass theorem.

Corollary 4. *Suppose $\mathcal{F}$ is a multi-layer functional ANN. Then for some real-valued continuous function $f : \mathbb{R} \to \mathbb{R}$, there exists a set of weights W such that $\forall \epsilon > 0$, $\|\mathcal{F}(\xi) - f\|_\infty < \epsilon$*

Proof. In this proof we will omit the inductive step as it is elementary and employs the same logic as the basis step. Consider the first neural layer

$$\sigma^{(1)}(j) = g\left(\int_{R^{(0)}} \xi(i) w^{(0)}(i, j)\, di\right) \tag{10}$$

because we take $w^{(0)}$ to be some approximating polynomial by the Stone-Weierstrass theorem, let $w^{(0)} = \left[(g^{-1})' \circ (h(\Xi, j))\right] h'(\Xi, j)$ approximately, where Ξ is the primitive of ξ. Supposing that h is some polynomial satisfying $h(\Xi, j)\Big|_{R^{(0)}} = f(j)$, then by the chain rule of integration

$$\begin{aligned}\sigma^{(1)}(j) &= g\left(\int_{R^{(0)}} \xi(i) \left[(g^{-1})' \circ (h(\Xi, j))\right] h'(\Xi, j)\, di\right) \\ &= g\left(g^{-1}\left(h(\Xi, j)\right)\right)\Big|_{R^{(0)}} \\ &= f(j)\end{aligned} \tag{11}$$

□

At this point it is important to note that the proof given above implies that ξ is disregarded through manipulation of $w^{(0)}$. Instead, h is a function of Ξ which is the primitive of ξ. If we were to not let $h(\Xi, j)\Big|_{R^{(0)}} = f(j)$, then we could claim that for arbitrary h we have proven any functional composition of ξ is possible; that is, in some light sense we have proven Cybenko's theorem in the general case by treating the weight set as some hypersurface. Moreover, the intuition follows that if we were to discretized the satisfying h and ξ (Theorem 1) then it is possible that a similar weight surface is developed for a trained $\mathcal{N}$. This result is remarkable as new light is shed on the black-box model of neural networks showing that approximation of h is made in the discrete sense.

Although we have shown through the corollary that approximation of arbitrary functional composition is possible, we have yet to consider values of w in the general sense. In other words, what can be said about the general approximation of bounded linear operators mapping $C(\mathbb{R}^n)$ to $C(\mathbb{R}^n)$ where C denotes the set of continuous (integrable) real valued functions? Evidently, the form of $\mathcal{F}$ resembles the general class of integral transforms, $\int f(x) \cdot E(x, k)\, dx$. Integral transforms are shown to approximate a multitude of operators through varying kernel functions. For example consider some $g(t)$ and the integral transform

$$(\mathcal{P}g)(s) = \int_0^\infty g(t)\delta'(s - t)\, dt \tag{12}$$

where δ is the Dirac-Delta function. Then we have that $(\mathcal{P}g)(s) = \frac{dg}{dt}\Big|_s$ by the properties of the Delta function. Similar approximations of linear operators can be made by varying the kernel function E. Thus there is considerable interest in determining the density of integral transforms and thereby functional neural networks in the set of bounded linear operators.

Further investigation into dense forms of bounded linear operators leads us to the Schwartz theorem of bounded linear operator representation by integral kernels. The theorem simply states that all linear operators can be represented in some light sense by integral transforms with arbitrary kernels. However, this theorem is too general for our purposes and we would like to show that in the specific case of some functional neural network $\mathcal{F}$ that for any given layer such that $l \neq 0$ any linear operator can be approximated with point-wise convergence from the Weierstrass polynomial approximation theory.

In order to do this we will return to the Riesz representation theorem that states the following (Hartig, 1983).

Theorem 5. *Let $\phi : C(X) \to \mathbb{R}$ be any bounded linear form where X is a compact Hausdorff space and $C(X)$ is the Banach space of continuous functions over X. Then*

there exists a unique regular Borel measure μ on X such that

$$\phi(f) = \int_X f(t)\, d\mu(t), \quad f \in C(X), \quad t \in X \tag{13}$$

and $\|\phi\| = |\mu|(X)$ where $|\mu|$ is the total variation of μ on X.

As opposed to generalizing Cybenko's theorem to Banach spaces ($\mathbb{R}^\infty$), we can actually manipulate the representation theorem to encapsulate bounded linear operators over locally compact Hausdorf spaces. Using the aforementioned logic the universal representation theorem for functional neural networks is now proposed.

Theorem 6. *Given a functional neural network $\mathcal{F}$ then some layer $l \in \mathcal{F}$, the let $K : C(R^{(l)}) \to C(R^{(l)})$ be a bounded linear operator. If we denote the operation of layer l on layer $l-1$ as $\sigma^{(l+1)} = g\left(\Sigma_{l+1}\sigma^{(l)}\right)$, then for every $\epsilon > 0$, there exists a weight polynomial $w^{(l)}(i,j)$ such that the supremum norm over $R^{(l)}$*

$$\left\| K\sigma^{(l)} - \Sigma_{l+1}\sigma^{(l)} \right\|_\infty < \epsilon \tag{14}$$

Proof. Let $\zeta_t : C(R^{(l)}) \to R^{(l)}$ be a linear form which evaluates its arguments at $t \in R^{(l)}$; that is, $\zeta_t(f) = f(t)$. Then because ζ_t is bounded on its domain, $\zeta_t \circ K = K^*\zeta_t$ is a bounded linear functional. Then from the Riesz Representation Theorem (Theorem 1) we have that there is a unique regular Borel measure μ_t on $R^{(l)}$ such that

$$\left(K\sigma^{(l)}\right)(t) = K^*\zeta_t\left(\sigma^{(l)}\right) = \int_{R^{(l)}} \sigma^{(l)}(s)\, d\mu_t(s), \quad \|\mu_t\| = \|K^*\zeta_t\| \tag{15}$$

Then if there exists a regular Borel measure μ such that μ_t is significantly smaller that μ for all t, then we have that, by the Radon-Nikodim derivative, $d\mu_t(s) = K_t(s)d\mu(s)$ under the assumption that K_t is L^1 integrable over $R^{(l)}$ with the measure μ. Thus it follows that

$$\begin{aligned} K\left[\sigma^{(l)}\right](t) &= \int_{R^{(l)}} \sigma^{(l)}(s)K_t(s)\, d\mu(s) \\ &= \int_{R^{(l)}} \sigma^{(l)}(s)K(t,s)\, d\mu(s). \end{aligned} \tag{16}$$

Therefore, for any bounded linear operator $K : C(X) \to C(X)$ there exists a unique $K(t,s)$ such that $K[f] = \int_X f(s)K(t,s)d\mu(s)$. Now we show that the operation of Σ_l can approximate any such operator. Because K is of the form of Σ_l where the only difference is the weighting function, we assert the following claim.

Let G be defined as a linear functional applied to a Gaussian heat kernel whose application with a function $f : \mathbb{R} \to \mathbb{R}$ yields the following definition,

$$G[f](x) = \frac{1}{b\sqrt{\pi}} \int_{\mathbb{R}} f(u)e^{\left(\frac{u-x}{b}\right)}\, du. \tag{17}$$

Then it follows that by the Weierstrass approximation theorem that for all $\epsilon > 0$, the supremum norm $\|f - G[f]\|_\infty < \epsilon$. Then because G is a polynomial, f must be a limit of polynomials. So now consider the operation of $K\left[\sigma^{(l)}\right](t)$ with kernel $K(t,s)$. By the Weierstrass approximation theorem $K(t,s)$ must be a limit of polynomials and therefore we let $w^{(l)}(i,j)$ assume that limit. That is,

$$\begin{aligned} \lim_{b\to 0} \left\| K\left[\sigma^{(l)}\right] - \int_{R^{(l)}} \sigma^{(l)}(s)\, G[k]\, d\mu(s) \right\|_\infty = \\ \left\| K\left[\sigma^{(l)}\right] - \Sigma_{l+1}\left[\sigma^{(l)}\right] \right\|_\infty < \epsilon \end{aligned} \tag{18}$$

Thus we have that as a limit of polynomials the operation of any arbitrary layer of a functional neural network $\mathcal{F}$ approaches any arbitrary linear bounded operator $K : C(R^{(l)}) \to C(R^{(l)})$; that is, functionals of the form Σ_{l+1} are dense in the set of all bounded continuous linear operators. □

In the above proof, a remarkable fact has been shown: functional neural networks can perform most mathematical operations on their inputs, and so neural networks preserve universal approximation in infinite dimensions!

4. Generalized Artificial Neural Networks

To implement Functional Neural Networks in a meaningful context, we need some algorithm which uses the computational power of $\{\mathcal{F}\}$ to yield finite dimensional classifications of ξ.

So the question arises: *does there exist some more general structure which includes FNNs and ANNs without piecewise approximation?* The solution to such questions must furthermore maintain universal approximation and computational evaluability so as to allow the implementation of a real-world algorithm.

Therefore, in this section of the paper we will propose the Generalized Artificial Neural Network through the examination of specific operations on different layers and then search the space of these GANNs for an algorithm that lets us make use of $\{\mathcal{F}\}$.

4.1. The Generalization of ANNs

In order to generalize ANNs in a way that follows logically, we take the initial definition of functional neural networks and consider the notion of a layer.

Definition 7. *If A, B are (possibly distinct) Banach spaces*

over $\mathbb{R}$, *we say* $\mathcal{G} : A \to B$ *is a generalized neural network if and only if*

$$\begin{aligned} \mathcal{G} : \sigma^{(l+1)} &= g\left(T_l\left[\sigma^{(l)}\right] + \beta^{(l)}\right) \\ \sigma^{(0)} &= \xi \end{aligned} \tag{19}$$

for some input $\xi \in A$..

In (19), it is unclear how the form of T_l is restricted. It might be recalled that T_l takes a form similar to the operation of a layer $l+1$ on l in the proof of universal approximation for $\{\mathcal{F}\}$. Let us consider a few other such forms of T_l by first stating a formal definition.

Definition 8. *We say that* T_l *is the operation of a layer* $l+1$ *on* l *in some* $\mathcal{G}$ *if and only if for* $l = L-1$, $T : C \to B$ *with* C *the codomain of* $\sigma^{(0)}$ *and for* $l = 0$, $T_l : A \to D$ *where* D *is the domain of* $\sigma^{(1)}$.

Using the above definition it is now possible to construct different classes of T_l using ideas directly from the constructions of $\mathcal{N}$ and $\mathcal{F}$.

Definition 9. *We suggest several classes of* T_l *as follows*

- T_l *is said to be* $\mathfrak{f}$ *functional if and only if* =

$$\begin{aligned} T_l = \mathfrak{f} : C(R^{(l)}) &\to C(R^{(l+1)}) \\ \sigma &\mapsto \int_{R^{(l)}} \sigma(i) w^{(l)}(i,j)\, di. \end{aligned} \tag{20}$$

- T_l *is said to be* $\mathfrak{n}$ *discrete if and only if*

$$\begin{aligned} T_l = \mathfrak{n} : \mathbb{R}^n &\to \mathbb{R}^m \\ \vec{\sigma} &\mapsto \sum_j^m \vec{e}_j \sum_i^n \sigma_i w_{ij}^{(l)} \end{aligned} \tag{21}$$

  where $\vec{e}_j$ *denotes the* j^{th} *basis vector in* $\mathbb{R}^m$.

- T_l *is said to be* $\mathfrak{n}_1$ *transitional if and only if*

$$\begin{aligned} T_l = \mathfrak{n}_1 : \mathbb{R}^n &\to C(R^{(l+1)}) \\ \vec{\sigma} &\mapsto \sum_i^n \sigma_i w_i^{(l)}(j). \end{aligned} \tag{22}$$

- T_l *is said to be* $\mathfrak{n}_2$ *transitional if and only if*

$$\begin{aligned} T_l = \mathfrak{n}_2 : C(R^{(l)}) &\to \mathbb{R}^m \\ \sigma(i) &\mapsto \sum_j^m \vec{e}_j \int_{R^{(l)}} \sigma(i) w_j^{(l)}(i)\, di \end{aligned} \tag{23}$$

Remark 10. *Observe that without loss of generality* $\{G\} \supset \{F\} \cup \{N\}$; *that is, every* $\mathcal{N}$ *can be expressed in terms of* $\mathcal{G} : \mathfrak{n} \to \cdots \to \mathfrak{n}$, *and the same for every* $\mathcal{F}$.

4.2. Continuous Classifiers and Dimensionality Reduction

In the case that data is sampled from a continuous process over time, it is general practice to build a feature vector in dimensions identically equal to the number of samples n. Then with such a feature vector, a practitioner can build a network architecture to accommodate her learning task. Unfortunately, if the input is of high quality, this introduces high-dimensionality into the weight matrix and thereby any optimization algorithm chosen.

However, searching $\{\mathcal{G}\}$ space yields a theoretically better approach.

Definition 11. $\mathcal{G}$ *is said to be a continuous classifier network if it is defined such that*

$$\mathcal{G} : \mathfrak{f} \to \mathfrak{f} \to \ldots \mathfrak{f} \to \mathfrak{n}_2, \tag{24}$$

and for every $l < L-1$,

$$w^{(l)}(i,j) = \sum_b^{Z_Y^{(l)}} \sum_a^{Z_X^{(l)}} k_{a,b}^{(l)} i^a j^b. \tag{25}$$

In construction every single weight polynomial can capture lower-order properties of the weight surface with even less dimensionality than a typical piecewise weight surface (w_{ij}). In fact polynomials can best-fit a function of n discrete points in the worst case with n coefficients, but they typically perform better within some ϵ error. [Numerical Methods of Curve Fitting. By P. G. Guest, Philip George Guest. Page 349.] The key here is that the resolution of the input function ξ only affects the feed-forward step on the first $\mathfrak{f}$ layer, and not the number, n, of parameters $k_{a,b}$ in the model.

Suppose in some instance we have a weight polynomial on an $\mathfrak{n}_2$ with $M \in O(1)$ parameters, which has learned a model. Then the following theorem gives a direct comparison of an ANN performing the same task.

Theorem 12. *Let* $\mathcal{G}$ *be a GANN with only one* $\mathfrak{n}_2$ *transitional layer. If a continuous function, say* $f(t)$ *is sampled uniformly from* $t = 0$, *to* $t = N$, *such that* $x_n = f(n)$, *and if* $\mathcal{G}$ *has an input function which is piecewise linear with*

$$\xi = (x_{n+1} - x_n)(z - n) + x_n \tag{26}$$

for $n \le z < n+1$, *then there exist some discrete neural network* $\mathcal{N}$ *such that* $\mathcal{G}(\xi) = \mathcal{N}(\boldsymbol{x})$.

Proof. Recall that for the jth output neuron of a single layer discretized neural network,

$$\mathcal{N}_j(\boldsymbol{x}) = g\left(\sum_{i=1}^N w_{i,j} x_i + \beta\right). \tag{27}$$

Let this $\mathcal{N}$ compose the same sigmoid as the aforementioned $\mathfrak{n}_2$ transitional layer. We need only show that equivalence holds on the inside.

Because the definition of a network at the jth componet gives us

$$\mathcal{G}_j(\xi) = g\left(\mathfrak{n}_2[\xi] + \beta\right), \tag{28}$$

it follows that,

$$\begin{aligned}
\mathfrak{n}_2(\xi) &= \int_R \xi(t)\, u_j(t)\, dt \\
&= \sum_{\mathfrak{n}}^{N-1} \int_{\mathfrak{n}}^{\mathfrak{n}+1} \left((x_{\mathfrak{n}+1} - x_{\mathfrak{n}})(z - \mathfrak{n}) + x_{\mathfrak{n}}\right) u_j(z)\, dz \\
&= \sum_{\mathfrak{n}}^{N-1} (x_{\mathfrak{n}+1} - x_{\mathfrak{n}}) \\
&\qquad \times \int_{\mathfrak{n}}^{\mathfrak{n}+1} (z - \mathfrak{n}) \sum_{\mathfrak{m}}^{M} k_{\mathfrak{m},j} z^{\mathfrak{m}}\, dz \\
&\qquad + x_{\mathfrak{n}} \int_{\mathfrak{n}}^{\mathfrak{n}+1} u_j(z)\, dz \\
&= \sum_{\mathfrak{n}}^{N-1} (x_{\mathfrak{n}+1} - x_{\mathfrak{n}}) \\
&\qquad \times \sum_{\mathfrak{m}}^{M} k_{\mathfrak{m},j} \left[\frac{1}{\mathfrak{m}+2} z^{\mathfrak{m}+2} - \frac{\mathfrak{n} z^{\mathfrak{m}+1}}{\mathfrak{m}+1}\right]_{\mathfrak{n}}^{\mathfrak{n}+1} \\
&\qquad + \sum_{\mathfrak{m}}^{M} k_{\mathfrak{m},j} x_{\mathfrak{n}} \frac{1}{\mathfrak{m}+1} z^{\mathfrak{m}+1} \Big|_{\mathfrak{n}}^{\mathfrak{n}+1}
\end{aligned} \tag{29}$$

Now, let $V_{\mathfrak{n},j} = \sum_{\mathfrak{m}} k_{\mathfrak{m},j} \left[\frac{1}{\mathfrak{m}+2} z^{\mathfrak{m}+2} - \frac{\mathfrak{n} z^{\mathfrak{m}+1}}{\mathfrak{m}+1}\right]_{\mathfrak{n}}^{\mathfrak{n}+1}$ and $Q_{\mathfrak{n},j} = \sum_{\mathfrak{m}} k_{\mathfrak{m},j} \frac{1}{\mathfrak{m}+1} z^{\mathfrak{m}+1} \Big|_{\mathfrak{n}}^{\mathfrak{n}+1}$. We can now easily simplify (29) using the telescoping trick of summation.

$$\begin{aligned}
\mathfrak{n}_2(\xi) &= x_N V_{N-1,j} \\
&\qquad + \sum_{\mathfrak{n}=2}^{N-1} x_{\mathfrak{n}} \left(Q_{\mathfrak{n},j} - V_{\mathfrak{n},j} + V_{\mathfrak{n}-1,j}\right) \\
&\qquad + x_1 \left(Q_{1,j} - V1, j\right).
\end{aligned} \tag{30}$$

By simply letting $w_{1,j} = (Q_{1,j} - V_{1,j})$, $w_{\mathfrak{n},j} = (Q_{\mathfrak{n},j} - V_{\mathfrak{n},j} + V_{\mathfrak{n}-1,j})$, and $w_{N,j} = V_{N-1,j}$ the following relation is satisfied, $\mathfrak{n}_2(\xi) = \mathfrak{n}(\boldsymbol{x})$. Hence, $\mathcal{G}(\xi) = \mathcal{N}(\boldsymbol{x})$ and the proof is complete. □

Notice that in the last proof, we went from $M \in O(1)$ dimensions to $O(N)$ where N is the number of samples on the input signal ξ! So there is constant space dimensionality reduction when using the $\mathfrak{n}_2$, and it is not to hard to see this for f.

Algorithm 1 Feedforward Propagation on $\{\mathcal{F}\}$

Input: input function ξ
for $l \in \{0, \ldots, L-1\}$ **do**
  for $t \in Z_X^{(l)}$ **do**
    Calculate $I_t^{(l)} = \int_{R^{(l)}} \sigma^{(l)}(j_l) \jmath_l^t \, dj_l$.
  end for
  for $s \in Z_Y^{(l)}$ **do**
    Calculate $C_s^{(l)} = \sum_a^{Z_X^{(l)}} k_{a,s} I_a^{(l)}$.
  end for
  Memoize $\sigma^{(l+1)}(j) = g\left(\sum_b^{Z_Y^{(l)}} \jmath^b C_b^{(l)}\right)$.
end for
The output is given by $\mathcal{F}[\xi] = \sigma^{(L)}$.

Algorithm 2 Error Backpropagation

Input: input γ, desired δ, learning rate α, time t.
for $l \in \{0, \ldots, L\}$ **do**
  Calculate $\Psi^{(l)} = g'\left(\int_{R^{(l-1)}} \sigma^{(l-1)} w^{(l-1)}\, dj_{l-1}\right)$
end for
For every t, compute $\mathfrak{B}_{L,t}$ from from (32).
Update the output coefficient matrix $k_{x,y}^{(L-1)} - I_x^{(L-1)} \int_{R^{(L)}} \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{(L)} \jmath_L^y \, dj_L \to k_{x,y}^{(L-1)}$.
for $l = L-2$ **to** 0 **do**
  If it is null, compute and memoize $\mathfrak{B}_{l+2,t}$ from (32).
  Compute but do not store $\mathfrak{B}_{l+1} \in \mathbb{R}$.
  Compute $\frac{\partial E}{\partial k_{x,y}^{(l)}} = \mathfrak{B}_l$ from from (32).
  Update the weights on layer l: $k_{x,y}^{(l)}(t) \to k_{x,y}^{(l)}$
end for

5. Implementation

With these theoretical guarantees given for $\{\mathcal{G}\}$, the implementation of the feedforward and error backpropagation algorithms in this context is an essential next step. Since the derivations are too long to include here, we defer the reader to other supplemental materials.

Feedforward propagation is straight forward, and relies on memoizing functionals by using the separability of weight polynomials. Essentially, integration need only occur once to yield coefficients on power functions. See Algorithm 1.

For error backpropagation, we chose the most direct analogue for the loss function, in particular since we showed universal approximation using the C^∞ norm, the integral norm will converge.

Definition 13. *For a functional neural network $\mathcal{F}$ and a dataset $\{(\gamma_{\mathfrak{n}}(j), \delta_{\mathfrak{n}}(j))\}$ we say that the error for a given $\mathfrak{n}$ is defined by*

$$E = \frac{1}{2} \int_{R^{(L)}} \left(\mathcal{F}(\gamma_{\mathfrak{n}}) - \delta_{\mathfrak{n}}\right)^2 \, dj_L \tag{31}$$

Using this definition we take gradient with respect to the coefficients of the polynomials on each weight surface. Eventially we get a recurrence relation in the same way one might for discrete neaural networks.

$$
\begin{aligned}
\mathfrak{B}_{L,t} &= \int_{R^{(L)}} \sum_{b}^{Z_Y^{(L-1)}} k_{t,b}^{(L-1)} j_L^b \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{(L)} \, dj_L. \\
\mathfrak{B}_{s,t} &= \int_{R^{(s)}} \sum_{b}^{Z_Y^{(s-1)}} \sum_{a}^{Z_X^{(s)}} k_{t,b}^{(s-1)} j_s^{a+b} \Psi^{(s)} \mathfrak{B}_{s+1,a} \, dj_s. \\
\mathfrak{B}_{l+1} &= \int_{R^{(l+1)}} \sum_{a}^{Z_X^{(l+1)}} j_{l+1}^{a+y} \Psi^{(l+1)} \mathfrak{B}_{l+2,a} \, dj_{l+1}. \\
\frac{\partial E}{\partial k_{x,y}^{(l)}} = \mathfrak{B}_l &= \int_{R^{(l)}} j_l^x \sigma^{(l)} \mathfrak{B}_{l+1} \, dj_l.
\end{aligned}
\tag{32}
$$

where Ψ is defined as $g'(T[\sigma] + \beta)$. Using this recurrence relation, we can drastically reduce the time to update each weight by memoizing. That philosophy yields algorithm 2, and therefore we have completed the practical analohgues to these algorithms.

6. Conclusion

In this paper we first extended the standard ANN recurrence relation to infinite dimensional input and output spaces. In this context of this new algorithm, FNN, we proved two new universal approximation theorems. The proposition of functional neural networks lead to new insights into the black box model of traditional neural networks.

Functional networks are a logical generalization of the discrete neural network and therefore all theorems shown for traditional neural networks apply to piecewise functional neural networks. Furthermore the creation of homologous theorems for universal approximation provided a way to find a relationship between the weights of traditional neural networks. This suggests that the discrete weights of a normal artificial neural network can be transformed into continuous surfaces which approximate kernels satisfying the training dataset. We then showed that functional neural networks are also able to approximate bounded linear operators.

The desire to implement $\{\mathcal{F}\}$ in actual learning problems motivated the exploration of a new space of algorithms: $\{G\}$. This new space not only contains standard ANNs and FNNs but also similar extensions such as that proposed in (Roux & Bengio, 2007). We then showed that a subset of $\{G\}$ containg algorithms called continuous classifiers actually reduce the dimensionality of the learning problem when expressed in n_2 instead of n layers.

Finally we proposed computationally feasible error back-propagation and forward propagation algorithms (up to an approximation).

6.1. Future Work

Although we have shown that advantage of using different subset of $\{G\}$ for learning tasks, there is still much work to be done. In this paper we did not explore different classes of weight surfaces, some of which may provide better computational integrability. It was also suggested to us that $\{F\}$ may link kernel learning methods and deep learning. Lastly, it remains to be seen how the general class of $\{G\}$, especially continuous classifiers, can be applied in practice.

References

Burch, Carl. A survey of machine learning. A survey for the Pennsylvania Governor's School for the Sciences, 2001.

Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–3314, 1989.

Hartig, Donald G. The riesz representation theorem revisited. *The American Mathematical Monthly*, 90(4): pp. 277–280, 1983. ISSN 00029890. URL http://www.jstor.org/stable/2975760.

McCulloch, Warren S and Pitts, Walter. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, (4), 1943.

Neal, Radford M. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

Roux, Nicolas L and Bengio, Yoshua. Continuous neural networks. In *International Conference on Artificial Intelligence and Statistics*, pp. 404–411, 2007.

Generalized Artifical Neural Networks

Approximated by Weierstrass Polynomials

Guss, William
wguss@berkeley.edu

April 5, 2016

Abstract

NOTE: This is an old abstract for only the first part of the paper (FNNs). In this paper we consider the traditional model of feed-forward neural networks proposed in (McCulloch and Pitts, 1949), and using intuitions developed in (Neal, 1994) we propose a method generalizing discrete neural networks as follows. In the standardized case, neural mappings $\mathcal{N} : \mathbb{R}^n \to [0,1]^m$ have little meaning when $n \to \infty$. Thus we consider a new construction $\mathcal{F} : \mathcal{X} \to \mathcal{Y}$ where the domain and codomain of $\mathcal{N}$ become infinite dimensional Hilbert spaces, namely the set of quadratically Lebesgue integrable functions L^2 over a real interval E and $[0,1]$ respectively. The derivation of this construction is intuitively similar to that of Lebesgue integration; that is, $\sum_i \sigma_i w_{ij} \to \int_{E \subset \mathbb{R}} \sigma(i) w(i,j)\, d\mu(s)$.

After establishing a proper family of "functional neural networks" $\mathcal{F}$, we show that $\mathcal{N}$ are a specific class of functional neural networks under specific constraints. More specifically in our first lemma, we prove that $\mathcal{F} \equiv \mathcal{N}$ for piecewise constant weight functions $w(i,j)$. Having done so, we then attempt to find an analogue to Cybenko's theorem of universal approximation for neural networks. Firstly, we prove as a corollary of the Weierstrass approximation theorem, that $w(i,j)$ can approximate a function, $f : E \to [0,1]$, satisfying $\|\mathcal{F}\xi - f(\xi)\|_\infty \to 0$. As a byproduct of the proof, we also establish a closed-form definition for the satisfying $w(i,j)$ and thereby through our first lemma provide novel insight into the actual form of the weight matrix $[w_{ij}]$ for trained $\mathcal{N}$. Finally we propose a universal approximation theorem for functional neural networks; that is, we show through the Riesz Representation Theorem that $\mathcal{F}$ approximates any bounded linear operator on $\mathcal{X}$.

In conclusion, we create a practical analogue of the error-backpropagation algorithm, and implement functional neural networks using Simpsons rule. We suggest that functional neural networks represent an interesting opportunity for the implementation of machine learning systems modeling functional transformation.

Contents

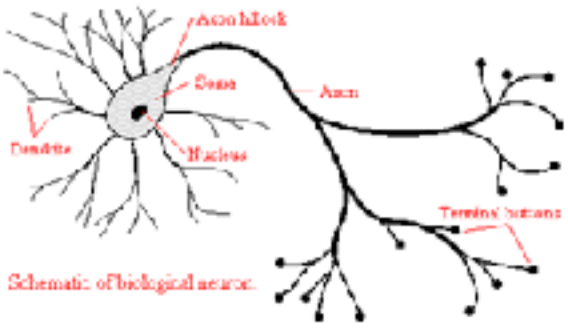


Figure 1: A biological neuron neuralimage.

1 Introduction

Machine learning is an emerging field that deals with the development of algorithms which can predict and classify novelties based on a set of prior intuitions mlsurvey. The field incorporates ideas from biology, computer science, numerical analysis, and statistics. In recent years machine learning has entered the main stream through web services like Google, Facebook, and Amazon. There is an incentive from both academia and industry to expand machine learning techniques and applications.

One of the most popular algorithms of the field is the artificial neural network (ANN). Although there are numerous mathematical interpretations of neural networks, we will primarily focus on the expansion of one type, feed-forward neural networks. As ANNs are based off the structure of biological neurons, a biological approach is necessary to understand this interpretation.

1.1 Biological Neuron

A single neuron consists of the cell body (the soma), the dendrites, and the axon. Mathematically we wish to examine the process of neural activation, the events which lead to the excitation of the axon. Consider a neuron that has activated anterior neurons (those which are connected dendritically); that is, the neuron is receiving input along all of its dendrites. These electrical inputs propagate through the dendrites and become integrated on the soma as electrical membrane potential griffith. The soma then acts as the primary computational unit and activates the axon when a threshold of input activity is reached. More specifically, when a membrane potential of about -60 mV is reached on the soma, the hillock zone, or axon hillock, activates the axon by applying proteins to an ion channel which creates action potential along the axon bioneuron.

1.2 Artificial Neurons

With this in mind it is now possible to construct a mathematical model of an artificial neuron. Let A_j, P_j be the set of anterior and posterior neurons of a neuron, j. Then, the cell membrane naturally becomes a linear combination of the dendritic potentials.

Definition 1.2.1. *We say that* net_j *is the net electric potential over the membrane, if for*

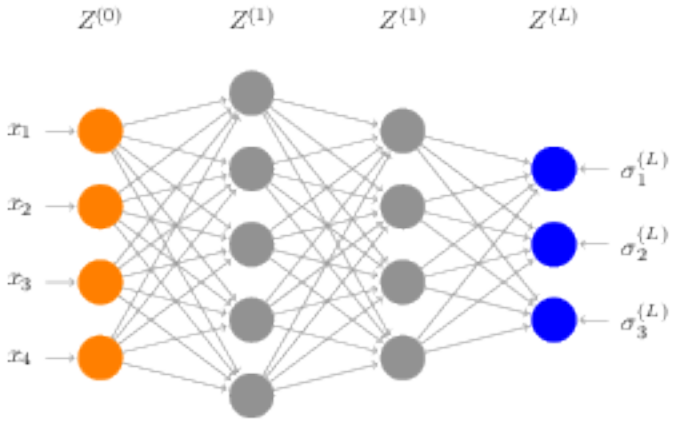


Figure 2: An example of a feed-forward ANN, $\mathcal{N}$ with four layers.

a natural neural resting potential β,

$$net_j = \sum_{i \in A_j} w_{ij}\sigma_i + \beta$$

where w_{ij} *is the dendritic connection strength from the* i^{th} *anterior neuron to* j, σ_i *is the action potential being propagated from the* i^{th} *anterior neuron.*

Furthermore, the thresholding of the hillock zone is given by some real valued sigmoidal function g bijective and differentiable over $\mathbb{R}$.

Definition 1.2.2. *We call* σ_j *the action potential of a neuron* j *if*

$$\sigma_j = g\left(\text{net}_j\right)$$

for some continuous real valued, monotonically increasing function g.

This model of the artificial neuron follows from the work that Pitt and McCulloch did in representing neural activity as logical thresholding elements mcculloch.

1.3 Feed-Forward Artificial Neural Networks

Now we have a sufficient mathematics base to define the feed-forward artificial neural network. The concept of a feed-forward ANN is biologically motivated by the functional organization of the visual cortex. It is appropriate to divide the structure of the visual cortex into layers which are denoted V1, V2, V3, and so on. The layers are organized such that a given layer is directly adjacent to and exhibiting full connectedness to the subsequent layer, an example being V1 to V2, V2 to V3, and subsequently for all of the primary layers of the visual cortex. From a functional point of view these layers store levels of visual abstraction like lines and shapes on the lower layers to faces and abstract visual concepts on the highest layersvisualcortex.

The goal is to model this representation of increasing abstraction whilst maintain adjacency and full topological connectedness. Thus we construct a set of neural layers with cardinality $L+1$, and connections as depicted in Figure 2.

Definition 1.3.1. *We say $\mathcal{N}$ is a feed-forward neural network if for an input vector $\boldsymbol{x}$,*

$$\begin{aligned}\mathcal{N} : \sigma_j^{(l+1)} &= g\left(\sum_{i \in Z^{(l)}} w_{ij}^{(l)} \sigma_i^{(l)} + \beta^{(l)}\right) \\ \sigma_j^{(1)} &= g\left(\sum_{i \in Z^{(0)}} w_{ij}^{(0)} x_i + \beta^{(0)}\right)\end{aligned}$$

Where $1 \leq l \leq L-1$.

For mathematical convenience let us denote $\sigma_j^{(l)}$ as the output of the j^{th} neuron on layer l. In this construction we prefer three different types of neurons, the input neuron, the hidden neuron, and the output neuron. In the case of the input neuron, there is no sigmoidal activation function, and instead we assign each $\sigma_j^{(0)}$ to a real value which is then weighted by the dendritic input strength of each anterior neuron. Moreover an input neuron only exists on the 0^{th} layer. In the case of each hidden layer we adopt the model described for the standard neuron as aforementioned where our sigmoid activation function $g = \tanh(\text{net})$ is the hyperbolic tangent.Finally, the output layer usually have a linear sigmoid activation as to achieve output scaling beyond $[1, -1]$ in the previous layers. Once again the output layer can only exist on the layer L.

1.4 Error Backpropagation

With the functional organization of the network complete, we now need to develop the notion of learning. For the purposes of this paper we will describe a gradient descent method for learning called error-backpropagation. In the mathematical model we find conveniently that the degrees of freedom are then the dendritic weights between any two neurons. Thus these weights must be optimized against some desired output. This leads to the following multi-dimensional error function.

Definition 1.4.1. *We call E the error function of a neural network $\mathcal{N}$, if for an input vector $\boldsymbol{x}$*

$$E\left(w_{00}^{(0)}, w_{01}^{(0)}, \ldots, w_{ij}^{(L)}\right) = \frac{1}{2} \sum_{i \in Z^{(L)}} \left(\sigma_i^{(L)} - \delta_i\right)^2$$

where $\boldsymbol{\delta}$ is some desired output vector corresponding to $\boldsymbol{x}$.

Then the goal is to optimize this error function such that a reasonable local minimum is found. We then choose to modify each weight in the direction of greatest decrease for the error function.

Definition 1.4.2. *We call $\boldsymbol{\nabla} E$ the gradient of E if*

$$\boldsymbol{\nabla} E = \left(\frac{\partial E}{\partial w_{00}^{(0)}}, \frac{\partial E}{\partial w_{01}^{(0)}}, \ldots, \frac{\partial E}{\partial w_{ij}^{(L)}}\right)$$

for all weights, $w_{ij}^{(l)}$, in feed-forward ANN $\mathcal{N}$.

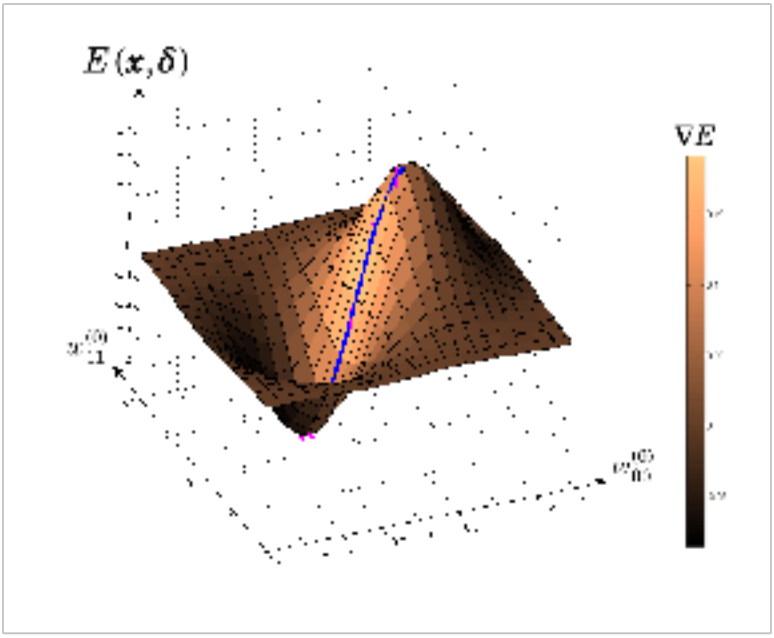


Figure 3: Gradient descent on $E(\boldsymbol{x},\boldsymbol{\delta})$ with descent path shown in blue.

Conveniently the gradient of a function describes a vector whose direction is the greatest increase of a function. Thus to optimize our weights so that the lowest error is achieved, we update the weights as follows: $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \alpha\boldsymbol{\nabla}E$ where alpha is some learning rate, a process which is depicted in Figure 3 rumelhart1988learning.

The calculation of each $\frac{\partial E}{\partial w_{ij}^{(L)}}$ is non-trivial given that each weight influences the error function in multiple ways. To find the contribution of a single weight, recall that every single neuron is connected to all neurons in the anterior and posterior layers. So, a single weight will influence not only its posterior neurons sigmoidal output but also that of every neuron for any path from the posterior neuron to the set of output neurons. Thus, the multiplicity of contribution via different neural routes follows directly from the multidimensional chain-

rule. Recall the differential operator $D_y f = \frac{\partial f}{\partial x}$,

$$\begin{aligned}
\frac{\partial E}{\partial w_{ij}^{(l)}} &= D_{\sigma_0^{(L)}} \cdot D_{\text{net}} \sigma_0^{(L)} \cdot D_{\sigma_0^{(L-1)}} \text{net} \cdot \ \cdots \ \cdot D_{\text{net}} \sigma_j^{(l+1)} \cdot D_{w_{ij}^{(l)}} \text{net} \\
&+ \ D_{\sigma_1^{(L)}} \cdot D_{\text{net}} \sigma_1^{(L)} \cdot D_{\sigma_0^{(L-1)}} \text{net} \cdot \ \cdots \ \cdot D_{\text{net}} \sigma_j^{(l+1)} \cdot D_{w_{ij}^{(l)}} \text{net} \\
&\ \ \vdots \\
&+ \ D_{\sigma_n^{(L)}} \cdot D_{\text{net}} \sigma_n^{(L)} \cdot D_{\sigma_0^{(L-1)}} \text{net} \cdot \ \cdots \ \cdot D_{\text{net}} \sigma_j^{(l+1)} \cdot D_{w_{ij}^{(l)}} \text{net} \\
&+ \ D_{\sigma_0^{(L)}} \cdot D_{\text{net}} \sigma_0^{(L)} \cdot D_{\sigma_1^{(L-1)}} \text{net} \cdot \ \cdots \ \cdot D_{\text{net}} \sigma_j^{(l+1)} \cdot D_{w_{ij}^{(l)}} \text{net} \\
&\ \ \vdots \\
&+ \ D_{\sigma_n^{(L)}} \cdot D_{\text{net}} \sigma_n^{(L)} \cdot D_{\sigma_1^{(L-1)}} \text{net} \cdot \ \cdots \ \cdot D_{\text{net}} \sigma_j^{(l+1)} \cdot D_{w_{ij}^{(l)}} \text{net} \\
&\ \ \vdots \\
&+ \ D_{\sigma_n^{(L)}} \cdot D_{\text{net}} \sigma_n^{(L)} \cdot D_{\sigma_{a_n}^{(L-1)}} \text{net} \cdot \ \cdots \ \cdot D_{\text{net}} \sigma_j^{(l+1)} \cdot D_{w_{ij}^{(l)}} \text{net} \\
&= \sum_{a_1}^{Z^{(L)}} \sum_{a_2}^{Z^{(L-1)}} \cdots \sum_{a_{a_n}}^{Z^{(l+2)}} \frac{\partial E}{\partial \sigma_{a_1}^{(L)}} \frac{\partial \sigma_{a_1}^{(L)}}{\partial \text{net}} \frac{\partial \text{net}}{\partial \sigma_{a_2}^{(L-1)}} \ \cdots \ \frac{\partial \sigma_{a_{a_n}}^{(l+2)}}{\partial \text{net}} \frac{\partial \text{net}}{\partial \sigma_j^{(l+1)}} \frac{\partial \sigma_j^{(l+1)}}{\partial \text{net}} \frac{\partial \text{net}}{\partial w_{ij}^{(l)}}
\end{aligned} \tag{1.4.1}$$

At first sight this algorithm looks quite complicated, but a computational implementation would be able to cache certain sums and essentially reduce the complexity thereof. With the error backpropagation algorithm complete, the notion of an artificial neural network, its training, and processing is complete. Now it is possible to conjecture on variants thereof and present the primary scope of this essay.

1.5 The Research Question

This construction is of serious mathematical interest as feed-forward neural networks have been shown to be universal approximators; that is, they can approximate any $f : A \to B$, where $A, B \in \mathbb{R}^m$ are vector spaces. However it is not certain what information the approximation provides: the gradient descent algorithm does not reveal any connections between the values of the inputs. A standing question in the field asks what can be said about the weights satisfying an approximation of f besides that they exist. Therefore it is of considerable interest to explore the general form of the weights as training may not be necessary and computational complexity can be lowered.

It is the subject of this research essay to explore the neural networks through a mathematical exploration. To do this, a technique from economic mathematics is employed. It is typical that to analyze a discrete economic model, time is considered continuous and summations become integrals. To investigate neural networks then, this technique will be applied. Thus the question arises: **what can be said about artificial neural networks as the number of nodes approaches infinity and how can a real valued, continuous analogue for neural networks contribute to or aid in understanding the black box model of artificial neural networks?**

In this paper we will generalize the notion of the universal approximation for arbitrary

vector space mappings to arbitrary approximation of any $f : L^1(\mathbb{R}^n) \to C^\infty(\mathbb{R}^n)$ by examining the structure of feed-forward ANNs as the number of nodes for each layer becomes uncountably bounded in $\mathbb{R}^n$. Such a generalization requires that a continuum of neural components be made, and that a continuous weight tensor or hypersurface must exist in order to maintain the topological connectedness as prescribed by the discrete model.

2 Functional Neural Networks

2.1 Derivation

Now that the functional neural network has been theoretically defined, the next step is to determine whether the implementation of a numerical algorithm is possible. The problem is approached using techniques already developed for discretized networks. To produce an output for a network with L layers, the implementation would propagate through all the layers while caching previous computations along the way. Likewise, for the error back-propagation method, we will cache and eliminate variables until we are able to define an algorithm that is suitably computable.

In order to begin exploration of such an implementation, we need to both more rigorously define our notion of the weight surface and what is itself computationally evaluable.

As were for discretized neural networks, the parameters of functional neural networks are defined. In particular, the weight surface for each layer is ideally optimized for a desired output operator. To construct these weight surfaces we will parameterize them using coefficients of Weierstrass polynomials. Consider the following definition.

Definition 2.1.1. *We say that a weight function is parameterized if it is defined as follows.*

$$w^{(l)}(i,j) = \sum_{x_{2l+1}}^{Z_Y^{(l)}} \sum_{x_{2l}}^{Z_X^{(l)}} k^{(l)}_{x_{2l},x_{2l+1}} j_l^{x_{2l}} j_{l+1}^{x_{2l+1}} \tag{2.1.1}$$

where $k_{a,b}$ is some real valued coefficient and the order of the polynomial is arbitrary.

With that in mind, let us define the notion of numerical integrability (and thereby evaluability).

Definition 2.1.2. *Let $f : \mathbb{R}^{2n} \to \mathbb{R}$, We say that some function ϕ of the form*

$$\phi(x) = \int_{E \subset \mathbb{R}^n} f(x,y)dy \tag{2.1.2}$$

can be numerical integrated if and only if

$$\phi(x) = h(x) \int_{E \subset \mathbb{R}^n} g(y)dy \tag{2.1.3}$$

for some $h, g : \mathbb{R}^n \to \mathbb{R}$.

Using the above definition we will explore the feed-forward and back-propagation actions of any given $\mathcal{F}$.

2.1.1 Feed-Forward Propagation

Consider the notion of numerical integrability defined in definition ?? applied to the calculation of output for some functional neural network.

Theorem 2.1.3. *If $\mathcal{F}$ is a functional neural network with L consecutive layers, then given any l such that $0 \leq l < L$, $\sigma^{(l+1)}$ is numerically integrable, and if ξ is any continuous and Riemann integrable input function, then $\mathcal{F}[\xi]$ is numerically integrable.*

Proof. Consider the first layer. We can write the sigmoidal output of the $(l+1)^{\text{th}}$ layer as a function of the previous layer; that is,

$$\sigma^{(l+1)} = g\left(\int_{R^{(l)}} w^{(l)}(j_l, j_{l+1})\sigma^{(l)}(j_l)\, dj_l\right). \tag{2.1.4}$$

Clearly this composition can be expanded using the polynomial definition of the weight surface. Hence

$$\begin{aligned}\sigma^{(l+1)} &= g\left(\int_{R^{(l)}} \sigma^{(l)}(j_l) \sum_{x_{2l+1}}^{Z_Y^{(l)}} \sum_{x_{2l}}^{Z_X^{(l)}} k_{x_{2l},x_{2l+1}} j_l^{x_{2l}}\, j_{l+1}^{x_{2l+1}}\, dj_l\right) \\ &= g\left(\sum_{x_{2l+1}}^{Z_Y^{(l)}} j^{x_{2l+1}} \sum_{x_{2l}}^{Z_X^{(l)}} k_{x_{2l},x_{2l+1}} \int_{R^{(l)}} \sigma^{(l)}(j_l) j_l^{x_{2l}}\, dj_l\right),\end{aligned} \tag{2.1.5}$$

and therefore $\sigma^{(l+1)}$ is numerically integrable. For the purpose of constructing an algorithm, let $I_{x_{2l}}^{(l)}$ be the evaluation of the integral in the above definition for any given x_{2l}

It is important to note that the previous proof requires that $\sigma^{(l)}$ be Riemann integrable. Hence, with ξ satisfying those conditions it follows that every $\sigma^{(l)}$ is integrable inductively. That is, because $\sigma^{(0)}$ is integrable it follows that by the numerical integrability of all l, $\mathcal{F}[\xi] = \sigma^{(L)}$ is numerically integrable. This completes the proof. □

Using the logic of the previous proof, it follows that the development of some inductive algorithm is possible.

2.1.2 Feed-Forward Algorithm

Fortunately in the proof it was shown that by calculation of a constant, I, on each layer, the functional neural network becomes numerically integrable. The mechanism by which this can occur leads us to a simple algorithm for calculating $\mathcal{F}[\xi]$:

1. For each $l \in \{0, \ldots, L-1\}$
   (a) For all $t \in Z_X^{(l)}$, calculate
   $$I_t^{(l)} = \int_{R^{(l)}} \sigma^{(l)}(j_l) j_l^t\, dj_l. \tag{2.1.6}$$
   (b) Calculate, for every $s \in Z_Y^{(l)}$,
   $$C_s^{(l)} = \sum_{x_{2l}}^{Z_X^{(l)}} k_{x_{2l},s} I_{x_{2l}}^{(l)} \tag{2.1.7}$$
   (c) Finally, using (??) and (??), cache
   $$\sigma^{(l+1)} = g\left(\sum_{x_{2l+1}}^{Z_Y^{(l)}} j^{x_{2l+1}} C_{x_{2l+1}}^{(l)}\right) \tag{2.1.8}$$
   for use in the next iteration of loop.
2. The last $\sigma^{(l)}$ calculated is the output of the functional neural network.

2.1.3 Continuous Error Backpropagation

Just as important as the feed-forward of neural network algorithms is the notion of training. As is common with many non-convex problems with discretized neural networks, a stochastic gradient descent method will be developed using a continuous analogue to error backpropagation.

As is typical in optimization, a loss function is defined as follows.

Definition 2.1.4. *For a functional neural network $\mathcal{F}$ and a dataset $\{(\gamma_n(j), \delta_n(j))\}$ we say that the error for a given n is defined by*

$$E = \frac{1}{2} \int_{R^{(L)}} \left(\mathcal{F}(\gamma_n) - \delta_n\right)^2 \, dj_L \tag{2.1.9}$$

This error definition follows from $\mathcal{N}$ as the typical error function for $\mathcal{N}$ is just the square norm of the difference of the desired and predicted output vectors. In this case we use the L^2 norm on $C(R^{(L)})$ in the same fashion.

We first propose the following lemma as to aid in our derivation of a computationally suitable error backpropagation algorithm.

Lemma 2.1.5. *Given some layer, $l > 0$, in $\mathcal{F}$, functions of the form $\Psi^{(l)} = g'\left(\Sigma_l \sigma^{(l)}\right)$ are numerically integrable.*

Proof. If

$$\Psi^{(l)} = g'\left(\int_{R^{(l-1)}} \sigma^{(l-1)} w^{(l-1)} \, dj_{l-1}\right) \tag{2.1.10}$$

then

$$\Psi^{(l)} = g'\left(\sum_b^{Z_Y^{(l-1)}} j_l^b \sum_a^{Z_X^{(l-1)}} k_{a,b}^{(l-1)} \int_{R^{(l-1)}} \sigma^{(l-1)} j_{l-1}^a \, dj_{l-2}\right) \tag{2.1.11}$$

hence Ψ can be numerically integrated and thereby evaluated. □

The ability to simplify the derivative of the output of each layer greatly reduces the computational time of the error backpropagation. It becomes a function defined on the interval of integration of the next iterated integral.

Theorem 2.1.6. *The gradient, $\nabla E(\gamma, \delta)$, for the error function (77) on some $\mathcal{F}$ can be evaluated numerically.*

Proof. Recall that E over $\mathcal{F}$ is composed of $k_{x,y}^{(l)}$ for $x \in Z_X^{(l)}, y \in Z_Y^{(l)}$, and $0 \leq l \leq L$. If we show that $\frac{\partial E}{\partial k_{x,y}^{(l)}}$ can be numerically evaluated for arbitrary, l, x, y, then every component of ∇E is numerically evaluable and hence ∇E can be numerically evaluated. Given some arbitrary l in $\mathcal{F}$, let $n = L - l$. We will examine the particular partial derivative for the case that $n = 1$, and then for arbitrary n, induct over each iterated integral.

Consider the following expansion for $n = 1$,

$$\begin{aligned}
\frac{\partial E}{\partial k_{x,y}^{(L-n)}} &= \frac{\partial}{\partial k_{x,y}^{(L-1)}} \frac{1}{2} \int_{R^{(l)}} \left[\mathcal{F}(\gamma) - \delta\right]^2 \, dj_L \\
&= \int_{R^{(p)}} \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{(L)} \int_{R^{(l-1)}} j_{L-1}^x j_L^y \sigma^{(L-1)} dj_{L-1} \, dj_L \\
&= \int_{R^{(p)}} \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{(L)} j_L^y \int_{R^{(l-1)}} j_{L-1}^x \sigma^{(L-1)} dj_{L-1} \, dj_L
\end{aligned} \tag{2.1.12}$$

Since the second integral in (??) is exactly $I_x^{(L-1)}$ from (??), it follows that

$$\frac{\partial E}{\partial k_{x,y}^{(n)}} = I_x^{(L-1)} \int_{R^{(l)}} \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{(L)} j_L^y \, dj_L \tag{2.1.13}$$

and clearly for the case of $n = 1$, the theorem holds.

Now we will show that this is all the case for larger n. It will become clear why we have chosen to include $n = 1$ in the proof upon expansion of the pratial derivative in these higher order cases.

Let us expand the gradient for $n \in \{2, \ldots, L\}$.

$$\frac{\partial E}{\partial k_{x,y}^{L-n}} = \int_{R^{(L)}} \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{(L)} \underbrace{\int_{R^{(L-1)}} w^{(L-1)} \Psi^{(L-1)} \int \ldots \int_{R^{(L-n+1)}} w^{(L-n+1)} \Psi^{(L-n+1)}}_{n-1 \text{ iterated integrals}}$$
$$\int_{R^{(L-n)}} \sigma^{(L-n)} j_{L-n}^{a} j_{L-n+1}^{b} \, dj_{L-n} \ldots dj_L \tag{2.1.14}$$

As aforementioned, proving the $n = 1$ case is required because for $n = 1$, (??) has a section of $n - 1 = 0$ iterated integrals which cannot be possible for the proceeding logic.

We now use the order invariance properly of iterated integrals (that is, $\int_A \int_B f(x,y) \, dxdy = \int_B \int_A f(x,y) \, dydx$) and reverse the order of integration of (??).

In order to reverse the order of integration we must ensure each iterated integral has an integrand which contains variables which are guaranteed integration over some region. To examine this, we propose the following recurrence relation for the gradient.

Let $\{B_s\}$ be defined along $L - n \leq s \leq L$, as follows

$$\begin{aligned} B_L &= \int_{R^{(L)}} \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{(L)} B_{L-1} \, dj_L, \\ B_s &= \int_{R^{(l)}} \Psi^{(l)} \sum_{a}^{Z_X^{(l)}} \sum_{b}^{Z_Y^{(l)}} j_l^a j_{l+1}^b B_{l-1} \, dj_l, \\ B_{L-n} &= \int_{R^{(L-n)}} j_{L-n}^x j_{L-n+1}^y \, dj_{L-n} \end{aligned} \tag{2.1.15}$$

such that $\frac{\partial E}{\partial k_{x,y}^{(l)}} = B_L$. If we wish to reverse the order of integration, we must find a reoccurrence relation on a sequence, $\{\mathfrak{B}_s\}$ such that $\frac{\partial E}{\partial k_{x,y}^{(L-n)}} = \mathfrak{B}_{L-n} = B_L$. Consider the gradual reversal of (??).

Clearly,

$$\frac{\partial E}{\partial k_{x,y}^{(l)}} = \int_{R^{(L-n)}} \sigma^{(L-n)} j_{L-n}^{x} \int_{R^{(L)}} \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{L} \int_{R^{(L-1)}} w^{(L-1)} \Psi^{(L-1)}$$
$$\int \ldots \int_{R^{(L-n+1)}} j_{L-n+1}^{y} w^{(L-n+1)} \Psi^{(L-n+1)} \, dj_{L-n+1} \ldots dj_L dj_{L-n} \tag{2.1.16}$$

is the first order reversal of (??). We now show the second order case with first weight

function expanded.

$$\frac{\partial E}{\partial k^{(l)}_{x,y}} = \int_{R^{(L-s)}} \sigma^{(L-n)} j^x_{L-n} \int_{R^{(L-s+1)}} \sum_b^{Z_Y} \sum_a^{Z_X} k_{a,b} j^{a+y}_{L-n+1} \Psi^{(L-n+1)} \int_{R^{(L)}} [\mathcal{F}(\gamma) - \delta] \Psi^L$$
$$\int \cdots \int_{R^{(L-s+1)}} j^b_{L-n+2} \mathfrak{U}^{(L-n+2)} \Psi^{(L-n+2)} \, dj_{L-n+1} \ldots dj_L dj_{L-n}. \tag{2.1.17}$$

Repeated iteration of the method seen in (??) and (??), where the inner most integral is moved to the outside of the $(L-s)^{\text{th}}$ iterated integral, with s is the iteration, yields the following full reversal of (??). For notational simplicity recall that $l = L - n$, then

$$\frac{\partial E}{\partial k^{(l)}_{x,y}} = \int_{R^{(l)}} \sigma^{(l)} j^x_l \int_{R^{(l+1)}} \sum_a^{Z_X^{(l+1)}} j^{a+y}_{l+1} \Psi^{(l+1)} \int_{R^{(l+2)}} \sum_b^{Z_Y^{(l+1)}} \sum_c^{Z_X^{(l+2)}} k^{(l+1)}_{a,b} j^{b+c}_{l+2} \Psi^{(l+2)}$$
$$\int_{R^{(l+3)}} \sum_d^{Z_Y^{(l+2)}} \sum_e^{Z_X^{(l+3)}} k^{(l+2)}_{c,d} j^{d+e}_{l+3} \Psi^{(l+3)} \int \cdots \int_{R^{(L)}} \sum_q^{Z_Y^{(L-1)}} k^{(L-1)}_{p,q} j^q_L \, [\mathcal{F}(\gamma) - \delta] \Psi^{(L)}$$
$$dj_L \ldots dj_{L-n}. \tag{2.1.18}$$

Observing the reversal in (??), we yield the following recurrence relation for $\{\mathfrak{A}_s\}$. Bare in mind, $l = L - n$, x and y still correspond with $\frac{\partial E}{\partial k^{(l)}_{x,y}}$, and the following relation uses its definition on s for cases not otherwise defined.

$$\begin{aligned}
\mathfrak{A}_{L,t} &= \int_{R^{(L)}} \sum_b^{Z_Y^{(L-1)}} k^{(L-1)}_{t,b} j^b_L \, [\mathcal{F}(\gamma) - \delta] \, \Psi^{(L)} \, dj_L. \\
\mathfrak{A}_{s,t} &= \int_{R^{(s)}} \sum_b^{Z_Y^{(s-1)}} \sum_a^{Z_X^{(s)}} k^{(s-1)}_{t,b} j^{a+b}_s \Psi^{(s)} \mathfrak{A}_{s+1,a} \, dj_s. \\
\mathfrak{A}_{l+1} &= \int_{R^{(l+1)}} \sum_a^{Z_X^{(l+1)}} j^{a+y}_{l+1} \Psi^{(l+1)} \mathfrak{A}_{l+2,a} \, dj_{l+1}. \\
\frac{\partial E}{\partial k^{(l)}_{x,y}} = \mathfrak{A}_l &= \int_{R^{(l)}} j^x_l \sigma^{(l)} \mathfrak{A}_{l+1} \, dj_l.
\end{aligned} \tag{2.1.19}$$

Note that $\mathfrak{A}_{L-n} = B_L$ by this logic.

With (??), we need only show that $\mathfrak{A}_{L-n}$ is integrable. Hence we induct on $L-n \leq s \leq L$ over $\{\mathfrak{A}_s\}$ under the proposition that $\mathfrak{A}_s$ is not only numerically integrable but also constant.

Consider the base case $s = L$. For every t, because every function in the integrand of $\mathfrak{A}_L$ in (??) is composed of j_L, functions of the form $\mathfrak{A}_L$ must be numerically integrable and clearly, $\mathfrak{A}_L \in \mathbb{R}$.

Now suppose that $\mathfrak{A}_{s+1,t}$ is numerically integrable and constant. Then, trivially, $\mathfrak{A}_{s,u}$ is also numerically integrable by the contents of the integrand in (??) and $\mathfrak{A}_{s,u} \in \mathbb{R}$. Hence, the proposition that $s+1$ implies s holds for $l+1 < s < L$.

Lastly we must show that both $\mathfrak{A}_{l+1}$ and $\mathfrak{A}_l$ are numerically integrable. By induction $\mathfrak{A}_{l+2}$ must be numerically integrable. Hence by the contents of its integrand $\mathfrak{A}_{l+1}$ must

also be numerically integrable and real. As a result, $\mathfrak{A}_l = \frac{\partial E}{\partial k_{x,y}^{(l)}}$ is real and numerically integrable.

Since we have shown that $\frac{\partial E}{\partial k_{x,y}^{(l)}}$ is numerically integrable, ∇E must therefore be numerically evaluable as aforementioned. This completes the proof. □

2.1.4 Continuous Error Backpropagation Algorithm

The logic of the proceeding proof required the establishment of $\{\mathfrak{A}_s\}$ as a sequence with a recurrence relation defining each component of the gradient on some coefficient in the network. Interestingly enough for L large enough certain elements of $\{\mathfrak{A}_s\}$ can be reused for different n. In order to more accurately describe this process, we propose the following algorithm for calculating gradients in stochastic gradient descent.

1. For all $\lambda \in \{1, \dots, L\}$, calculate $\Psi^{(\lambda)}$.
2. Calculate $K - \nabla E \to K$ where K is the vector of all coefficient matrices by calculating each partial matrix over $l \in \{0, \dots, L-1\}$
   (a) If $l = L - 1 \Leftrightarrow (n = 1)$, then store into the coefficient matrix
   $$k_{x,y}^{(L-1)} - I_x^{(L-1)} \int_{R^{(L)}} \left[\mathcal{F}(\gamma) - \delta\right] \Psi^{(L)} j_L^y \, dj_L \to k_{x,y}^{(L-1)} \tag{2.1.20}$$
   for every x, y.
   (b) If $l < L - 1 \Leftrightarrow n > 1$, then
      i. Ensure that $\mathfrak{A}_{l+2,t}$ from (??) is computed for all $t \in Z_X^{(l+1)}$
      ii. Then for all $x \in Z_X^{(l)}$ and $y \in Z_Y^{(l)}$,
         A. Compute
         $$\mathfrak{A}_{l+1} = \int_{R^{(l+1)}} \sum_{a}^{Z_X^{(l+1)}} j_{l+1}^{a+y} \Psi^{(l+1)} \mathfrak{A}_{l+2,a} \, dj_{l+1}. \tag{2.1.21}$$
         B. Compute
         $$\frac{\partial E}{\partial k_{x,y}^{(l)}} = \mathfrak{A}_l = \int_{R^{(l)}} j_l^x \sigma^{(l)} \mathfrak{A}_{l+1} \, dj_l. \tag{2.1.22}$$
         C. Update the weights such that
         $$k_{x,y}^{(l)} - \mathfrak{A}_l \to k_{x,y}^{(l)} \tag{2.1.23}$$

With the completion of the implementation, the theoretical definition of functional neural networks is complete.

3 Generalized Artificial Neural Networks

We have clearly demonstrated the theoretical power of FNNs, but does there exist some more general structure which includes FNNs and ANNs without piecewise approximation? In other words, can we determine a form which encompasses the discrete classification of continuous datasets or visa versa all the while maintains either the power of discrete or functional artificial neural networks? The solution to such questions must furthermore maintain universal approximation and computational evaluability so as to allow the implementation of a real-world algorithm.

In this section of the paper we will propose the Generalized Artificial Neural Network through the examination of specific operations on different layers.

3.1 The Generalization of ANNs

In order to generalize ANNs in a way that follows logically, we take the initial definition of functional neural networks and consider the notion of a layer.

Definition 3.1.1. *If A, B are (possibly distinct) Hilbert spaces over $\mathbb{R}$, we say $\mathcal{G} : A \to B$ is a generalized neural network if and only if*

$$\begin{aligned} \mathcal{G} : \sigma^{(l+1)} &= g\left(T_l\left[\sigma^{(l)}\right] + \beta^{(l)}\right) \\ \sigma^{(0)} &= \xi \end{aligned} \tag{3.1.1}$$

for some input $\xi \in A$.

In (??), it is unclear how the form of T_l is restricted. It might be recalled that T_l takes a form similar to the operation of a layer $l+1$ on l in the proof of universal approximation for $\{\mathcal{F}\}$. Let us consider a few other such forms of T_l by first stating a formal definition.

Definition 3.1.2. *We say that T_l is the operation of a layer $l+1$ on l in some $\mathcal{G}$ if and only if for $l = L-1$, $T : C \to B$ with C the codomain of $\sigma^{(0)}$ and for $l = 0$, $T_l : A \to D$ where D is the domain of $\sigma^{(1)}$.*

Using the above definition it is now possible to construct different classes of T_l using ideas directly from the constructions of $\mathcal{N}$ and $\mathcal{F}$.

Definition 3.1.3. *We suggest several classes of T_l as follows*

- *T_l is said to be $\mathfrak{f}$ functional if and only if =*

$$\begin{aligned} T_l = \mathfrak{f} : C(R^{(l)}) &\to C(R^{(l+1)}) \\ \sigma &\mapsto \int_{R^{(l)}} \sigma(i) w^{(l)}(i,j)\ di. \end{aligned} \tag{3.1.2}$$

- *T_l is said to be $\mathfrak{n}$ discrete if and only if*

$$\begin{aligned} T_l = \mathfrak{n} : \mathbb{R}^n &\to \mathbb{R}^m \\ \vec{\sigma} &\mapsto \sum_j^m \vec{e_j} \sum_i^n \sigma_i w_{ij}^{(l)} \end{aligned} \tag{3.1.3}$$

  where $\vec{e_j}$ denotes the j^{th} basis vector in $\mathbb{R}^m$.

- *T_l is said to be π_1 transitional if and only if*

$$T_l = \pi_1 : \mathbb{R}^n \to C(R^{(l+1)}) \\ \vec{\sigma} \mapsto \sum_i^n \sigma_i w_i^{(l)}(j). \tag{3.1.4}$$

- *T_l is said to be π_2 transitional if and only if*

$$T_l = \pi_2 : C(R^{(l)}) \to \mathbb{R}^m \\ \sigma(i) \mapsto \sum_j^m \vec{e}_j \int_{R^{(l)}} \sigma(i) w_j^{(l)}(i)\, di \tag{3.1.5}$$

With the characterization of these layer classes complete, foundational theorems about this new generalization must be proposed. First we will show inclusion of other neural algorithms followed by universal approximation theorems (as by necessity).

Theorem 3.1.4. *If $\{\mathcal{G}\}$ is the set of all generalized artificial neural networks then $\{\mathcal{F}\} \cup \{\mathcal{N}\}$ is a subset of $\{G\}$.*

Proof. Since $\{\mathcal{N}\} \subset \{\mathcal{F}\}$, we need only show that $\{\mathcal{F}\} \subset \{\mathcal{G}\}$, but for the sake of the reader, consider that one might take any $\mathcal{N}$ and show equivalency with some $\mathcal{G}$ strictly composed of T_l which are discrete.

For the case of functional inclusion, consider any functional neural network $\mathcal{F}$ with L layers. Then let us construct $\mathcal{G} : C\left(R^{(0)}\right) \to C\left(R^{(L)}\right)$. First we endow $\mathcal{G}$ with T_l which are strictly functional. Let each T_l have an equivalent weight function $w^{(l)}$ to that of $\mathcal{F}$ on layer l. This construction yields the following recurrence relation:

$$\begin{aligned} \mathcal{G} : \sigma^{(l+1)}(j) &= g\left(\int_{R^{(l)}} \sigma^{(l)}(i) w^{(l)}(i,j)\, di + \beta^{(l)}\right) \\ \sigma^{(0)}(j) &= \xi(j), \end{aligned} \tag{3.1.6}$$

which is exactly equivalent to that of $\mathcal{F}$. Hence for any $\mathcal{F}$ there exists a corresponding $\mathcal{G}$ with equivalency. Thus $\{\mathcal{F}\} \subset \{\mathcal{G}\}$. □

3.2 Input Quality as a Kernal Predicate for Neural Networks

In this section we will explore various anylitically integrable input functions on certain $\mathcal{G}$. In particular, we wish to explore the case of π_2 as it seems this construction may embody the most practical application of this new generalization: the classification of continuous signals. We wish to prove mathemnatically and demonstrate through application that such a method is better than current classifciation methods for continuous signals, or at the least, the two methods are isomorphic.

It is well established that all processes are sampled discretely, so in the least, some sort of approximate method must be used to create suitable inputs ξ for π_2 transitional layers. The following exploration is interesting.

Theorem 3.2.1. *Let $\mathcal{G}$ be a GANN with only one π_2 transitional layer. If a continuous function, say $f(t)$ is sampled uniformly from $t = 0$, to $t = N$, such that $x_n = f(n)$, and if $\mathcal{G}$ has an input function which is piecewise linear with*

$$\xi = (x_{n+1} - x_n)(z - n) + x_n \tag{3.2.1}$$

for $n \leq z < n+1$, then there exist some discrete neural network $\mathcal{N}$ such that $\mathcal{G}(\xi) = \mathcal{N}(\boldsymbol{x})$.

Proof. Recall that for the jth output neuron of a single layer discretized neural network,

$$\mathcal{N}_j(\boldsymbol{x}) = g\left(\sum_{i=1}^{N} w_{i,j} x_i + \beta\right). \tag{3.2.2}$$

Let this $\mathcal{N}$ compose the same sigmoid as the aforementioned π_2 transitional layer. We need only show that quivalence holds on the inside.

Because

$$\mathcal{G}_j(\xi) = g\left(\pi_2[\xi] + \beta\right), \tag{3.2.3}$$

it follows that,

$$\begin{aligned}
\pi_2(\xi) &= \int_R \xi(t)\, u_j(t)\, dt \\
&= \sum_{n}^{N-1} \int_n^{n+1} \left((x_{n+1} - x_n)(z - n) + x_n\right) u_j(z)\, dz \\
&= \sum_{n}^{N-1} (x_{n+1} - x_n) \int_n^{n+1} (z - n) \sum_{m}^{M} k_{m,j} z^m\, dz + x_n \int_n^{n+1} u_j(z)\, dz \\
&= \sum_{n}^{N-1} (x_{n+1} - x_n) \sum_{m}^{M} k_{m,j} \left[\frac{1}{m+2} z^{m+2} - \frac{n z^{m+1}}{m+1}\right]_n^{n+1} + \sum_{m}^{M} k_{m,j} x_n \frac{1}{m+1} z^{m+1}\Big|_n^{n+1}
\end{aligned} \tag{3.2.4}$$

Now, let $V_{n,j} = \sum_m k_{m,j} \left[\frac{1}{m+2} z^{m+2} - \frac{n z^{m+1}}{m+1}\right]_n^{n+1}$ and $Q_{n,j} = \sum_m k_{m,j} \frac{1}{m+1} z^{m+1}\Big|_n^{n+1}$. We can now easily simplify (??) using the telescoping trick of summation. It follows that,

$$\pi_2(\xi) = x_N V_{N-1,j} + \sum_{n=2}^{N-1} x_n \left(Q_{n,j} - V_{n,j} + V_{n-1,j}\right) + x_1 \left(Q_{1,j} - V1, j\right). \tag{3.2.5}$$

By simply letting $w_{1,j} = (Q_{1,j} - V_{1,j})$, $w_{n,j} = (Q_{n,j} - V_{n,j} + V_{n-1,j})$, and $w_{N,j} = V_{N-1,j}$ the following relation is satisfied, $\pi_2(\xi) = \pi(\boldsymbol{x})$. Hence, $\mathcal{G}(\xi) = \mathcal{N}(\boldsymbol{x})$ and the proof is complete. □

The previous theorem shows that at the least there is an isomorphism from $\{\mathcal{G}\}$ of that kind to the discretized neural network for piecewise linear interpolations. What can be said about higher order interpolations? The following theorem provides the same isomorphism.

Theorem 3.2.2. *Let $\phi : \mathbb{R}^N \to C(\mathbb{R})$ be the mapping for the polynomial interpolation of a uniformly sampled function $f(t)$ described by some vector of points. Then if $\mathcal{G}$ is a GANN with only one π_2 transitional layer, then there exists a discretized neural network $\mathcal{N}$ such that for all $\boldsymbol{x} \in \mathbb{R}^N$,*

$$\mathcal{G}\left[\phi(\boldsymbol{x})\right] = \mathcal{N}(\boldsymbol{x}). \tag{3.2.6}$$

Proof. It has been well established that a closed form definition for polynomial interpolation is as follows. If N points are sampled from a function $f(t)$ along intervals $[p_i, p_{i+1}]$,

$$\phi(\boldsymbol{x}) = \sum_{n=0}^{N} \left(\prod_{\substack{0 \le m \le e \\ m \neq n}} \frac{z - p_m}{p_n - p_m} \right) x_n \,. \tag{3.2.7}$$

For simplicity, let $h_n(z) = \prod_m \frac{z-p_m}{p_n-p_m} = \sum_s H_{n,s} z^s$ for some $H_{n,s}$ which satisfy the relationship. Because (??) implies that only equivalence of π_2 and π must be shown,

$$\begin{aligned} \pi_2[\phi(\boldsymbol{x})] &= \int_R \phi(\boldsymbol{x}) \; u_j \; dz \\ &= \sum_n^N x_n \int_R h_n(z) u_j(z) \; dz \\ &= \sum_n^N x_n \sum_s^N H_{n,s} \sum_t^T k_{t,j} \frac{1}{s+t+1} z^{s+t+1} \Big|_R \end{aligned} \tag{3.2.8}$$

Now defining $w_{n,j} = \sum_s^N H_{n,s} \sum_t^T k_{t,j} \frac{1}{s+t+1} z^{s+t+1} \big|_R$, it holds that $\pi_2[\phi(\boldsymbol{x})] = \pi[\boldsymbol{x}]$. Therefore, $\mathcal{G}[\phi(\boldsymbol{x})] = \mathcal{N}(\boldsymbol{x})$. This completes the proof.

□

Considering the logic of the previous proof in reverse, it is clear the some neural networks might infact be approximating these π_2 transitional layers. The following is a theorem that suggests some discrete neural networks approximate kernels on infinite dimensional space.

Theorem 3.2.3. *Discrete nueral networks with one hidden layer can approximate a kernel $K(x, w)$ which induces the inner product on the infinite dimensional L_2 space of continuous functions.*

Proof. Consider the kernel, $K(x, w) = \langle \phi(x), \phi(w) \rangle_{L_2}$ where $\phi : \mathbb{R}^N \to L_2(\mathbb{R})$ as in (??). □

4 Conclusion

When the research question was posed, the goal was to develop a mathematical construct that is continuously homologous to standard and discrete artificial neural networks. This along with inspiration from common economic techniques like interpolation led us to consider a neural network structure containing infinite input, hidden, and output neurons. This new construction is named the functional neural network and has the same properties as discrete neural networks.

The proposition of functional neural networks lead to new insights into the black box model of traditional neural networks. First, functional networks are a logical generalization of the discrete neural network and therefore all theorems shown for traditional neural networks apply to piecewise functional neural networks. Furthermore the creation of homologous theorems for universal approximation provided a way to find a relationship between the weights of traditional neural networks. This suggests that the discrete weights of a normal artificial neural network can be transformed into continuous surfaces which approximate kernels satisfying the training dataset. Functional neural networks are also able to approximate bounded linear operators (the general form of functions), homologous to how ANNs apprxoimate functions. Finally a continuous version of the error backpropagation algorithm was developed, providing information into how the discrete error backprop algorithm operates: the chain rule just becomes convolution integration across partially derived anterior layers.

These algorithms, while theoretically feasible, were originally too computationally complex to have a practical application. The algorithms were reapproached mathematically to make feed forward and error backpropagation numerically integrable. This property is shown through the expansion of the weight polynomial and the interchanging of iterated integrals. From this, a computational implementation of the new generalized construct, FNNs, is created. Furthermore, the process of caching important values in weight coefficient calculation greatly reduced the original factorial complexity such that the algorithms could run in reasonable computational time.

Thus, the functional neural network not only provides novel mathematical insights behind the traditional neural network algorithm, it also becomes a feasible new machine learning method. This new field has many potential directions especially with continued techniques from mathematical analysis.

4.1 Future Work

The computational algorithm was applied to analyze the Laplace transform. However, there is still a range of applications to be explored. The original intent of this paper was to explore continuous data that exists in the real world. Sound waves and sight analysis by ears and eyes occurs on a continuous level yet most of the data analysis in these fields has discretized the data for numerical evaluation. The insertion of continuous sinusoidal definition of these datasets into a FNN would be a possible field of exploration.

On the theoretical side, it must be proven, or disproven, that functional neural networks are or are not analytically integrable in closed form. If they are integrable then the aforementioned computational implementation will be very simple and possibly faster than discrete neural networks. In the case that they are not, functional neural networks may remain simply a theoretical construct for understanding discrete neural networks. Evidence has been shown that it is integrable using a linear sigmoid activation functions. However,

for nonlinear sigmoid activation functions, there is a strong indication that there is no closed form interval solution.