Decay Makes Supervised Predictive Coding Generative

by

Wei Sun

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Predictive Coding is a hierarchical model of neural computation that approximates backpropagation using only local computations and local learning rules. An important aspect of Predictive Coding is the presence of feedback connections between layers. These feedback connections allow Predictive Coding networks to potentially be generative as well as discriminative. However, Predictive Coding networks trained on supervised classification tasks cannot generate accurate input samples close to the training inputs from the class vectors alone.

This problem arises from the fact that generating inputs from classes requires solving an underdetermined system, which contains an infinite number of solutions. Generating the correct inputs involves reaching a specific solution in that infinite solution space. But by imposing a minimum-norm constraint on the state nodes and the synaptic weights of a Predictive Coding network, the solution space collapses to a unique solution that is close to the training inputs. This minimum-norm constraint can be enforced by adding decay to the Predictive Coding equations.

Decay is implemented in the form of weight decay and activity decay. Analyses done on linear Predictive Coding networks show that applying weight decay during training helps the network learn weights that can generate the correct input samples from the class vectors, while applying activity decay during input generation helps to lower the variance in the network's generated samples. Additionally, weight decay regularizes the values of the network weights, avoiding extreme values, and improves the rate at which the network converges to equilibrium by regularizing the eigenvalues of the Jacobian at the equilibrium.

Experiments on the MNIST dataset of handwritten digits provide evidence that decay makes Predictive Coding networks generative even when the network contains deep layers and uses nonlinear tanh activations. A Predictive Coding network equipped with weight and activity decay successfully generates images resembling MNIST digits from the class vectors alone.

Acknowledgements

I would like to thank my supervisor, Jeff Orchard, for offering me the opportunity to work on such an interesting research project and for continuously providing me with support and direction throughout my Master's. I would also like to thank Tyler Jackson for pointing out that multimodal datasets can be a problem for generative Predictive Coding, and for providing helpful advice and code throughout my time working on this project. I would additionally like to thank Bryan Tripp for allowing me to work on a project that led me to discover the benefit that decay brings to Predictive Coding in the discriminative mode. Lastly, I would like to thank Yu Gu for helping me to understand the mathematics behind dynamical systems theory.

I would like to gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

Dedication

I dedicate this thesis to Mark Luo, for his enthusiasm in neural networks, and to Alan Zhang, for his inspiring play style in the social deception game Avalon. I also dedicate this thesis to my supervisor, Jeff Orchard, for the knowledge and motivation he has given me in learning about neuroscience. Thank you all for supporting me through my research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial neural networks (ANNs) have shown tremendous success at statistical learning due to the backpropagation learning algorithm [51]. In recent years, convolutional neural networks (CNNs) [35] trained by backpropagation have achieved widespread success at high-dimensional classification problems such as image recognition [29]. Generative adversarial networks have also attained success in generating images from a latent space [18], but not from a class vector alone.

Why might it be useful to generate a sample from a class vector? One reason is that we may want to have a network that maps bidirectionally between particular inputs and targets of a dataset. Such a network can be used as a part of other architectures in order to guarantee that a segment of that architecture produces unique, invertible data mappings. Another reason is that the neural network theoretically encodes a generative model, and so having the network generate a sample from a class verifies the efficacy of that model.

In this thesis, I will show that the regularizing effect of **decay** decisively benefits the process of generating salient inputs from class vectors in a type of network called **Supervised Predictive Coding**. Using decay, we can design a neural network that is both discriminative and generative, meaning that the network can *discriminate* the class of an input and *generate* valid input samples from the class using the same set of weights.

First, we will take a look at how backpropagation works in feed-forward neural networks, a basic type of ANN.

1.1 Artificial Neural Networks

Feed-forward neural networks are functions that map an input to an output. The network is a directed graph that consists of multiple hierarchical layers [51]. An example of an artificial neural network architecture is shown in Fig. 1.1.

The layer which receives the input to the function is called the input layer, while the layer that produces the output of the function is called the output layer. Intermediate layers between the input and output are called hidden or deep layers. Each layer consists of neurons, which store artificial firing rates called *activities*. In this thesis, the term *values* will be used to refer to the numerical representations of these activities.

In normal feed-forward neural networks, every neuron within a layer is connected to every neuron within the next layer. The strength of a connection between two neurons is called a *weight*, and the full matrix of connections between two adjacent layers of neurons is called a *weight matrix* whose elements are termed *weights*. Activities entering a layer

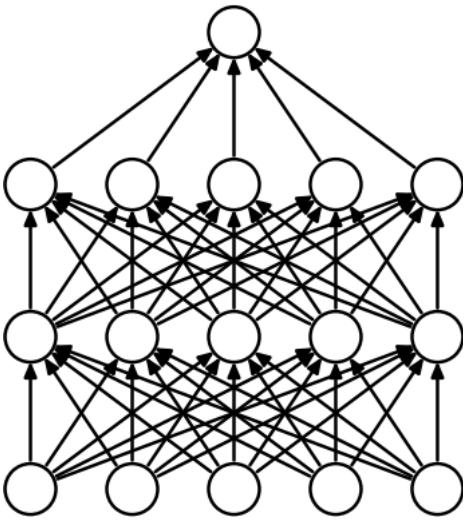


Figure 1.1: Standard feed-forward neural network, taken from [64]. The input layer is at the bottom, the output layer is the single neuron at the top, and intermediate layers are called deep layers.

can be called an *input current*.

A bias b can be added to the current. The bias is often treated as additional weights projecting from an extra neuron with constant value 1 in a preceding layer to every neuron in the next layer.

The current arriving at a receiving layer is transformed through a function f called an *activation*. The purpose of activations is to introduce nonlinearity into the network. Nonlinearities allow a neural network to perform more complex processing, such as separating an input into multiple classes where each class occupies nonlinearly separable space. Sample activation functions are tanh, arctan, the logistic function σ, and the rectified linear

unit, ReLU.

$$\text{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{1.1}$$

Denote $f^{(l)}$ to be the activation function for layer (l). Let $w_{i,j}^{(l-1)}$ be the strength of the synaptic weight connection between neuron j on layer $(l-1)$ and neuron i on layer (l). The feed-forward operation from a given layer of values $x^{(l-1)}$ with $n^{(l-1)}$ neurons to a receiving neuron $x_i^{(l)}$ on layer l, can be summarized as

$$x_i^{(l)} = f^{(l)} \left(\sum_{j=1}^{n^{(l-1)}} w_{i,j}^{(l-1)} x_j^{(l-1)} + b_i^{(l)} \right) \tag{1.2}$$

This equation can be simplified by placing every variable in matrix-vector form. Here, $W^{(l-1)}$ denotes the weight matrix connecting layers $(l-1)$ and (l), $\vec{x}^{(l-1)}$ denotes a vector of values from layer $(l-1)$, and $\vec{b}^{(l)}$ denotes a vector of biases added to the current entering layer (l),

$$\vec{x}^{(l)} = f^{(l)} \left(W^{(l-1)} \vec{x}^{(l-1)} + \vec{b}^{(l)} \right) \tag{1.3}$$

A feed-forward pass is to run equation (1.3) on an input sample or batch x through every layer of the neural network, producing the output y at the output layer.

The weights and biases of a feed-forward neural network are parameters to be learned. The number of layers, the number of neurons in each layer, and the activation function used at each layer are called *hyperparameters* which are chosen when the neural network is initialized. Another hyperparameter, the *learning rate*, governs the magnitude of each modification to the network's weights during learning.

Learning the weights and biases of a feed-forward neural network is achieved using an algorithm called **backpropagation** [51]. This algorithm is used for supervised learning, the setting in which the target output for a given input is known. Backpropagation additionally requires a cost function, C, also known as the objective or the loss, to quantify the difference between the outputs y produced by the neural network and the targets t. Popular neural network loss functions are *mean squared error* for continuous valued outputs and *cross entropy* for binary vector classification, also known as one-hot classification. Indexing each input-target pair in a dataset with i, and denoting y_i as the output corresponding to target t_i, cross entropy is

$$C(y,t) = -\sum_i t_i \log y_i + (1-t_i)\log(1-y_i) \tag{1.4}$$

Backpropagation finds the gradient of the cost function with respect to each weight and bias of the network. For the current z entering the output layer of the network, that is, the values projecting into the output layer before being transformed by the activation, the derivative of the loss C with respect to each component of the current z_i is

$$\frac{\partial C}{\partial z_i} = \frac{\partial C}{\partial y_i}\frac{dy_i}{dz_i} \tag{1.5}$$

For certain combinations of output activations and loss functions, the derivative above simplifies to the difference $y_i - t_i$. Examples of such combinations are cross entropy loss with logistic activation and cross entropy with the softmax activation function (not discussed in this thesis).

Backpropagation acquires its name from the repeated application of chain rule to find the derivative of the loss with respect to values further down the layers of the network.

Suppose that for layer (l), we have found $\frac{\partial C}{\partial z_j^{(l)}}$ for all j. Denote the values of the neurons at layer $(l-1)$ as $x^{(l-1)}$. Then we can also find $\frac{\partial C}{\partial z_i^{(l-1)}}$,

$$\frac{\partial C}{\partial z_i^{(l-1)}} = \sum_{j=1}^{n^{(l)}} \frac{dx_i^{(l-1)}}{dz_i^{(l-1)}} \frac{\partial z_j^{(l)}}{\partial x_i^{(l-1)}} \frac{\partial C}{\partial z_j^{(l)}} \tag{1.6}$$

We can simplify these terms. From (1.2), we can infer that, summing across j, $\frac{\partial z_j^{(l)}}{\partial x_i^{(l-1)}}$ simplifies to the column $w_{\cdot,i}^{(l-1)}$ of the weight matrix, while $\frac{dx_i^{(l-1)}}{dz_i^{(l-1)}}$ simplifies to the derivative of the activation function $f^{(l)}$ as long as the same activation is used for all neurons at layer (l). Therefore, treating $\frac{\partial C}{\partial z_j^{(l)}}$ and $\frac{\partial z_j^{(l)}}{\partial x_i^{(l-1)}}$ as vectors across j in equation (1.6) becomes

$$\frac{\partial C}{\partial z_i^{(l-1)}} = f'^{(l)}(x_i^{(l-1)}) w_{\cdot,i}^{(l-1)^T} \frac{\partial C}{\partial \bar{z}^{(l)}} \tag{1.7}$$

which in full matrix-vector form can be written as

$$\frac{\partial C}{\partial \bar{z}^{(l-1)}} = f'^{(l)}(\bar{x}^{(l-1)}) \odot W^{(l-1)^T} \frac{\partial C}{\partial \bar{z}^{(l)}} \tag{1.8}$$

Notice that $\odot$ is used in (1.8) to indicate a Hadamard product.

Given $\frac{\partial C}{\partial z_j^{(l)}}$, the derivatives of the loss with respect to the weights $w_{j,i}^{(l-1)}$ and biases $b_j^{(l)}$ are

$$\frac{\partial C}{\partial w_{j,i}^{(l-1)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{j,i}^{(l-1)}} \tag{1.9}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} \tag{1.10}$$

The expression $\frac{\partial z_j^{(l)}}{\partial w_{j,i}^{(l-1)}}$ simplifies to $h_i^{(l-1)}$, and $\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}$ simplifies to 1. Thus the update equations can be written as

$$\frac{\partial C}{\partial w_{j,i}^{(l-1)}} = \frac{\partial C}{\partial z_j^{(l)}} h_i^{(l-1)} \tag{1.11}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \tag{1.12}$$

Each weight $w_{j,i}^{(l-1)}$ is then incremented in the opposite direction of the derivative $\frac{\partial C}{\partial w_{j,i}^{(l-1)}}$ with the aforementioned learning rate α, and similarly for the bias. That is, for all layers (l), backpropagation performs gradient descent on the weights $w_{j,i}^{(l-1)}$,

$$w_{j,i}^{(l-1)} \leftarrow w_{j,i}^{(l-1)} - \alpha \frac{\partial C}{\partial w_{j,i}^{(l-1)}} \tag{1.13}$$

and on the biases $b_j^{(l)}$,

$$b_j^{(l)} \leftarrow b_j^{(l)} - \alpha \frac{\partial C}{\partial b_j^{(l)}} \tag{1.14}$$

The full learning algorithm for artificial neural networks is presented in Algorithm 1. In the algorithm, the term *epoch* refers to one full pass over the training set, and the term *batch* refers to the fact that the inputs are often placed in batches of a user-chosen size. For each epoch, the batches are recreated at random from all the input-target pairs. Optionally, the algorithm can also evaluate the total loss of the network on the training set after each

epoch.

Algorithm 1: Backpropagation

```
Initialize weights and biases;
for each epoch do
    for each batch x,t in dataset do
        y = feed-forward(x);
        Backpropagation(y, t);
    end
end
```

1.2 Biological Plausibility of Backpropagation

Despite the success of backpropagation, ANNs still exhibit weaknesses compared to human intelligence, such as requiring thousands or even millions of data points in order to learn. Human intelligence learns quickly from much fewer inputs. Trained neural networks are also susceptible to adversarial examples that do not easily fool humans [1]. One popular direction of progress is thus to explore neural models that are consistent with the architectural and computational constraints of biological brains. I will not investigate the aforementioned weaknesses in this thesis, but I will study a more biologically plausible model of neural computation as an alternative to backpropagation.

What does it mean for a model to be biologically plausible? I will use the criteria defined by Whittington and Bogacz. A biologically plausible model satisfies four criteria: local computation, local plasticity, minimal external control, and plausible architecture

[68].

Local computation means that the computations performed by a neuron must involve only the values of adjacently connected neurons. Similarly, local plasticity means that the weights of the neural network must be modified only using the values of the two neurons whose connection strength the weight represents [68].

Minimal external control means the network must behave autonomously, performing both inference and learning with as little external control routing information in different ways at different times as possible [68].

Lastly, plausible architecture means the architecture of the neural network should be consistent with the connectivity constraints that exist in the brain, and particularly the neocortex [68].

Currently, artificial neural networks do not resemble biological neural networks because the ANN learning algorithm, backpropagation, lacks biological plausibility. To show why backpropagation is not biologically plausible, it is useful to imagine that a separate network of neurons is projecting back values from the output layer. These values correspond to the backpropagation derivative terms that are computed and used to update the feed-forward network's weights and biases.

First, backpropagation violates local computation because each derivative term relies on the values of derivatives computed from many layers back. Backpropagation violates local plasticity for the same reason, since the updates to the weights and biases of the network rely on non-locally computed derivatives. These violations follow inherently from the fact that backpropagation uses the chain rule, which requires values from across the

entire chain. It is questionable if the cerebral cortex can even compute equations at all [9].

Backpropagation also violates minimal external control, since it requires the network to have distinct feed-forward and backpropagation phases, as shown in Algorithm 1. This implies an exterior force is switching the network between its two phases, which is implausible since all known brains are believed to be self-controlled.

Lastly, backpropagation violates plausible architecture through its use of symmetric weights [37]. In equation (1.8), the existence of the term $W^{(l-1)^T}$ implies that an entire set of synaptic connection strengths must be copied onto the aforementioned separate network projecting back backpropagation derivative terms [19]. This is known as the weight transport problem [37].

Since backpropagation violates all four biological plausibility criteria, artificial neural networks trained with backpropagation are biologically implausible. Why, then, does backpropagation work so well as a learning algorithm?

In the next chapter, we will investigate another neural framework called Predictive Coding that can approximate backpropagation learning [68]. Predictive Coding addresses some, though not all, of the biological plausibility issues of backpropagation, and it offers some interesting insights into the potential ways that biological neurons can perform learning and computation.

My thesis aims to show that making Predictive Coding networks generate inputs from class vectors requires solving a system that contains infinite solutions, and decay collapses the solution space to the one that resembles the input. Using Predictive Coding, we will create a neural network that can classify inputs and generate an input sample from classes.

Chapter 2

Predictive Coding

Predictive Coding (PC) is a theory of cortical computations posited to explain information processing in the brain. It is characterized by one-to-one pairings of *state* nodes and *error* nodes at each layer of a hierarchical neural network [47]. State nodes encode values much like the activities in artificial neural networks, and error neurons encode the differences between the predicted values projected down from the higher layers of the network and the values of sensory input projected up from lower layers. Predictive Coding networks also have forward and feedback connections between layers [47]. Forward connections are the weights that map values from a given layer to the next layer, while feedback connections are the weights that map values from an upper layer to the layer below.

Predictive Coding in the brain comes from the idea that the brain encodes the causes of sensory input as parameters of a generative model, and new sensory inputs are represented in terms of those causes [59]. Equipped with this generative model, the brain may be able

to perform inference on its environment. For example, our eyes perceive an image which is full of red and yellow light wavelengths, and our brain infers that it is looking at a sunset. But another possible inference is that it is looking at an autumn tree. In either case, the brain is making a prediction about what it is looking at.

Multiple different objects can give rise to the same image. Inferring the causes behind an outcome, such as the object behind an image, is an inverse problem that tends to be ill-posed, with multiple or zero solutions [59]. These problems require additional constraints in order to solve, and Predictive Coding may be able to provide these constraints [59, 7, 8, 24, 47, 58].

Rao and Ballard introduced Predictive Coding in the visual cortex as a hierarchical generative model [47]. An image seen by the brain is represented as a linear combination of basis vectors. These basis vectors are neural activities, and they are linearly projected to subsequent layers. The result of the multiplication can also be processed through an activation function such as *logistic*. The final neural activities can be thought of as a set of *causes* that together give rise to the image perceived by the brain. These activities are the *state nodes* in Predictive Coding.

Rao and Ballard proposed a hierarchical neural model in which every layer represents a set of causes. Each layer is multiplied by a synaptic weight matrix to generate another set of causes at the layer below, until the input image is generated at the input layer. In other words, each layer **predicts** the values of the activities of the layer below.

The model seeks to minimize the difference between the predictions and the actual activities at each layer. By encoding this difference in *error nodes* that exist at every layer,

and by performing gradient descent on the values of all the errors, the model becomes able to make accurate predictions. Gradient descent can be performed by modifying the values of each layer's state nodes as well as by modifying the synaptic weights connecting each layer.

Another conception of Predictive Coding is the PC/BC-DIM model by Spratling [59, 55]. BC stands for Biased Competition and proposes that visual stimuli compete to be represented by cortical activity [56]. This competition is achieved through excitatory feed-forward and feedback connections between populations of neurons, along with inhibitory lateral connections within each population [56, 10, 50].

Spratling proposes a reconciliation of biased competition and Predictive Coding [56]. Instead of excitatory feedback connections, feedback from higher layers inhibits the inputs to a neuron [22]. The usual way to represent the inhibitory effect of feedback is to have error nodes encode the difference between the state nodes they are directly connected to and the feedback that they receive from the layer above. Spratling's PC/BC-DIM model instead uses Divisive Input Modulation [62], in which errors are calculated using division instead of subtraction [59]. Error nodes encode the ratio between the state nodes they are connected to and higher layer's non-zero predictions sent down.

Many other implementations of Predictive Coding exist. Some network architectures have leveraged the principle of Predictive Coding–that is, the existence of predictions and errors–to perform specific vision-related tasks such as video prediction [38]. Predictive Coding can also be used in conjunction with other neural architectures, such as LSTMs and convolutional layers [38].

2.1 Predictive Coding with Free Energy Principle

We now return to this question: how does the brain solve the problem of inferring the causes of sensory input? Friston's Free Energy Principle explains that the brain solves this problem using a process called minimizing the brain's Free Energy [13]. According to Friston, Free Energy is an information theoretic quantity that bounds the evidence for a model of data, where data is sensory inputs and the model is encoded by the brain [14]. Free Energy is inspired by statistical mechanics and can be thought of as the amount of "entropy" in the brain. In other words, the process of minimizing Free Energy is the process of minimizing entropy. But what does it mean to minimize entropy in the brain?

First, the motivation for Free Energy rests upon the fact that self-organizing biological agents resist a tendency to disorder [14]. What this means is that biological organisms maintain their order by restricting themselves to a limited number of states [2], and Friston's proposal is that these states encode a generative model that predicts the conditions of the organism's environment. Entropy, which can be thought of as disorder, is therefore a misprediction of the environment, which can be defined as "surprise". Thus, minimizing Free Energy in the brain corresponds to minimizing the surprise experienced by the brain.

According to the Free Energy Principle, the introduction of sensory stimuli induces surprise in the brain. Cortical responses can be seen as the attempt to minimize the Free Energy induced by that stimulus and thereby encode the most likely cause of that stimulus [13]. Learning emerges from synaptic changes that minimize Free Energy, averaged over all stimuli encountered [13]. This statistical view of the brain forms the basis of the mathematics of the Free Energy Principle conception of Predictive Coding.

Bogacz presents an effective tutorial of the Free Energy Principle, explaining how it can be mapped onto a population of neurons [4]. This section follows and summarizes his tutorial.

Bogacz starts by considering an organism with some visual capability that attempts to infer the diameter v of a food item based on the light intensity it observes. Let u be a noisy estimate of the light intensity its visual receptor receives and let g denote a non-linear function that relates the average light intensity of an object with the object's diameter [4]. For example, $g(v) = v^2$. Next, assume that the perceived light intensity is normally distributed with mean $g(v)$ and variance Σ_u. Then the probability of observing u given v is

$$p(u|v) = \mathcal{N}(u; g(v), \Sigma_u) \tag{2.1}$$

The organism is also equipped with prior knowledge on how large food items are that it learnt from experience [4]. Assume that the animal expects this size to be normally distributed with mean v_p and variance Σ_p.

$$p(v) = \mathcal{N}(v; v_p, \Sigma_p) \tag{2.2}$$

When the animal has observed a particular value of light intensity u, it is then faced with the problem of estimating the diameter v of the food item, which can be interpreted as computing $p(v|u)$ [4]. It is possible to use Bayes' theorem to compute

$$p(v|u) = \frac{p(v)p(u|v)}{p(u)} \tag{2.3}$$

$$p(u) = \int p(v)p(u|v)dv \tag{2.4}$$

Bogacz notes that it would be difficult for an organism to perform this computation for two reasons. First, if g is a non-linear function, then $p(v|u)$ will not necessarily have a normal spread of values. Thus, representing $p(v|u)$ requires representing infinitely many values for each possible u rather than just representing summary statistics like mean and variance [4]. Second, it is hard to imagine a neural circuit that can compute (2.4), which is a normalization term that ensures all $p(v|u)$ integrate to 1. Bogacz suggests that the basal ganglia, a region of the brain implicated in reward-motivated decision making, can compute the normalization term for discrete probability distributions [5], but computation of the normalization for continuous distributions requires evaluating the integral [4]. This would be too hard for a simple neural circuit to do.

Bogacz then proposes that the brain seeks the most likely size of the food item v that maximizes $p(v|u)$. Denoting this size ϕ, the brain then attempts to find the ϕ that maximizes the probability density distribution $p(\phi|u)$. Since $p(\phi|u)$ depends on a ratio with $p(u)$ in the denominator and $p(u)$ does not depend on ϕ, we can just find ϕ that maximizes the terms $p(\phi)$ and $p(u|\phi)$ in the numerator [4]. Denote by F the logarithm of the numerator,

$$F = \ln p(\phi) + \ln p(u|\phi) \tag{2.5}$$

Since the natural logarithm function is monotonic, we can maximize the numerator by maximizing F. But why do we not just maximize the numerator directly? It is because $p(\phi)$ and $p(u|\phi)$ are normal distributions, and so the formula that represents them contains exponentiation terms. Taking the logarithm of those terms removes the exponentiation,

allowing a simpler derivation of the objective function that maximizes F [4].

$$\begin{aligned}
F &= \ln p(\phi) + \ln p(u|\phi) \\
&= \ln f(\phi; v_p, \Sigma_p) + \ln f(u; g(\phi), \Sigma_u) \\
&= \ln \left[\frac{1}{\sqrt{2\pi\Sigma_p}} e^{-\frac{(\phi - v_p)^2}{2\Sigma_p}} \right] + \ln \left[\frac{1}{\sqrt{2\pi\Sigma_u}} e^{-\frac{(u - g(\phi)^2}{2\Sigma_u}} \right] \\
&= \ln \frac{1}{\sqrt{2\pi}} - \frac{1}{2} \ln \Sigma_p - \frac{(\phi - v_p)^2}{2\Sigma_p} + \ln \frac{1}{\sqrt{2\pi}} - \frac{1}{2} \ln \Sigma_u - \frac{(u - g(\phi))^2}{2\Sigma_u} \\
&= \frac{1}{2} \left(- \ln \Sigma_p - \frac{(\phi - v_p)^2}{\Sigma_p} - \ln \Sigma_u - \frac{(u - g(\phi))^2}{\Sigma_u} \right) + C
\end{aligned}$$

Here, Bogacz incorporates all constant terms into a constant C, leaving only the terms containing variables we are concerned with for maximizing F. Then, taking the derivative of F with respect to ϕ [4] gets

$$\frac{\partial F}{\partial \phi} = \frac{v_p - \phi}{\Sigma_p} + \frac{u - g(\phi)}{\Sigma_u} \frac{\partial g}{\partial \phi} \tag{2.6}$$

Now, the computational process of finding ϕ that maximizes F can be expressed in terms of a differential equation of the variable ϕ with respect to time t,

$$\frac{d\phi}{dt} = \frac{v_p - \phi}{\Sigma_p} + \frac{u - g(\phi)}{\Sigma_u} \frac{\partial g}{\partial \phi} \tag{2.7}$$

Here, $\frac{d\phi}{dt}$ is simply set to $\frac{\partial F}{\partial \phi}$. The equation can be simulated with an appropriate Euler time step to arrive at an approximate ϕ that maximizes F, and in this manner, acquire the most likely size v of the food item of interest. But how could a network of neurons implement such a differential equation?

We can bring back the concept of error-encoding neurons discussed in earlier conceptions of Predictive Coding, and denote two of the terms in the above equation as error terms,

$$\varepsilon_p = \frac{\phi - v_p}{\Sigma_p} \tag{2.8}$$

$$\varepsilon_u = \frac{u - g(\phi)}{\Sigma_u} \tag{2.9}$$

Substituting these terms into (2.7) and replacing $\frac{\partial g}{\partial \phi}$ with $g'(\phi)$ gives

$$\frac{d\phi}{dt} = \varepsilon_u g'(\phi) - \varepsilon_p \tag{2.10}$$

Bogacz assumes that v_p, Σ_p, and Σ_u are encoded in the strengths of the synaptic connections since they represent prior knowledge, while ϕ, ε_u, and ε_p are encoded in the activity of neurons or neural populations. Next, Bogacz derives differential equations for ε_u and ε_p with respect to time [4],

$$\frac{d\varepsilon_p}{dt} = \phi - v_p - \Sigma_p \varepsilon_p \tag{2.11}$$

$$\frac{d\varepsilon_u}{dt} = u - g(\phi) - \Sigma_u \varepsilon_u \tag{2.12}$$

These two equations allow each error node to converge to the values defined in (2.8) and (2.9). This can be shown by setting (2.11) and (2.12) to 0. Figure 2.1 shows the architecture of the Predictive Coding network that facilitates equations (2.10), (2.11), and (2.12).

It is also possible to find ϕ that maximizes F by changing the terms v_p, Σ_p, and Σ_u, whose values are encoded in interlayer synaptic connections. This process can be interpreted as "learning". As the organism experiences more and more light intensity-food

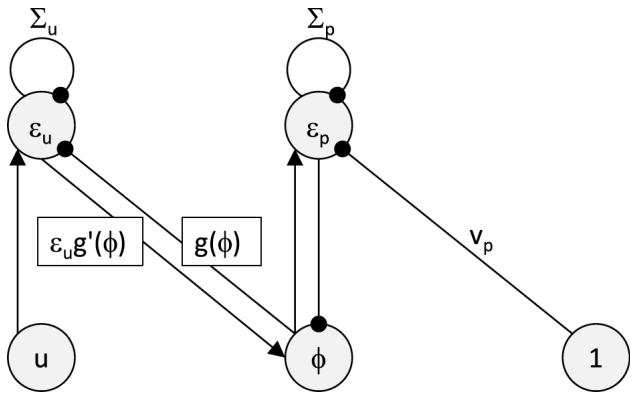


Figure 2.1: Predictive Coding neural network, taken from [4]. Circles denote neural nodes, arrows indicate excitatory connections, and lines ending with circles indicate inhibitory connections. Labels above connections denote their connection strengths and lack of label indicates a connection strength of 1. The rectangles indicate the values that need to be transmitted via the connections they label [4].

size relations, its knowledge, or "priors", should change to reflect those experiences as well. Bogacz derives updates for each of these terms,

$$\frac{\partial F}{\partial v_p} = \frac{\phi - v_p}{\Sigma_p} \tag{2.13}$$

$$\frac{\partial F}{\partial \Sigma_p} = \frac{1}{2}\left(\frac{(\phi - v_p)^2}{\Sigma_p^2} - \frac{1}{\Sigma_p}\right) \tag{2.14}$$

$$\frac{\partial F}{\partial \Sigma_u} = \frac{1}{2}\left(\frac{(u - g(\phi))^2}{\Sigma_u^2} - \frac{1}{\Sigma_u}\right) \tag{2.15}$$

Notice that equation (2.13) can be written as just $\frac{\partial F}{\partial v_p} = \varepsilon_p$. This implies that the update to the synaptic connection v_p is Hebbian because it depends only on the activity of presynaptic and postsynaptic neurons [4]. The other two equations for Σ_p and Σ_u can also be simplified similarly. In this thesis, we will ignore changes to Σ_p and Σ_u and focus on changes to v_p

as the main mechanism of learning.

Next, we can consider the function g. Bogacz conjectures that g is a function of two parameters, θ and v, such that $g(v, \theta) = \theta h(v)$, where $h(v)$ is some nonlinear function. Then we can derive the following update equations by substituting θ and h into $g(\phi)$,

$$\frac{d\phi}{dt} = \theta \varepsilon_u h'(\phi) - \varepsilon_p \tag{2.16}$$

$$\frac{d\varepsilon_p}{dt} = \phi - v_p - \Sigma_p \varepsilon_p \tag{2.17}$$

$$\frac{d\varepsilon_u}{dt} = u - \theta h(\phi) - \Sigma_u \varepsilon_u \tag{2.18}$$

The update to v_p can now be written as just an update to θ since $h(\phi)$ is an activation function that does not change [4],

$$\frac{\partial F}{\partial \theta} = \varepsilon_u h(\phi) \tag{2.19}$$

A Predictive Coding network that facilitates equations (2.16), (2.17), (2.18), and (2.19) is shown in Fig. 2.2.

Everything discussed thus far involves a single dimensional input, u. We can now scale up the dimensionality of the input and also the dimensionality of our network by allowing there to be multiple neurons per layer and also multiple layers.

The same objective function F can be used and derived for a vector of inputs and a vector of nodes at each layer of the neural network. Bogacz shows these derivations rigorously in his tutorial [4].

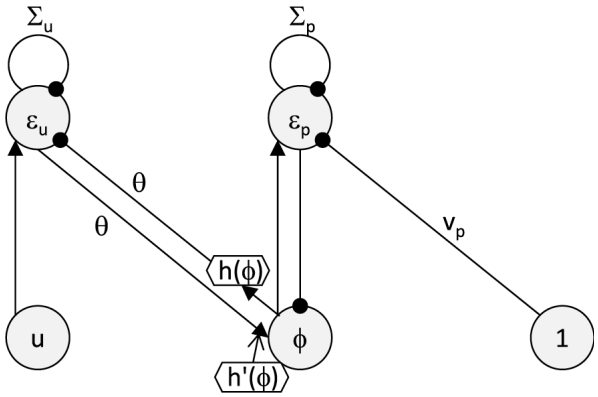


Figure 2.2: Predictive Coding neural network with $h(\phi)$, taken from [4]. Notice that connection weight strengths between ε_u and ϕ are now a scalar mapping θ followed by a nonlinear activation h or h'.

We now presume a vector $\bar{u}$ of inputs. Let our Predictive Coding model be a hierarchical neural network of n layers, where the input $\bar{u}$ is at layer 0 and a hypothetical *output* is at layer $n-1$. For each layer i between the input and output, a vector of error nodes $\bar{\varepsilon}^{(i)}$ connects to a vector of *states*, $\bar{\phi}^{(i)}$, with connection strength 1 on the same layer. For the interlayer connections, $\bar{\varepsilon}^{(i)}$ connects to $\bar{\phi}^{(i+1)}$ nodes on the next layer with connection weight strengths $\theta^{(i)}$. The activation function for layer i is $h^{(i)}()$.

How does this network relate to our previous, unidimensional network? Now that we have introduced hierarchy, we are not limited to just two sets of error nodes, ε_u and ε_p, but rather, we have as many vectors of error nodes $\bar{\varepsilon}^{(i)}$ as there are layers i. Further, we have as many internal state nodes $\bar{\phi}^{(i)}$ and interlayer connection weight strengths $\theta^{(i)}$ as we require as well. Our update equations become [4]

$$\frac{d\phi^{(i)}}{dt} = -\bar{\varepsilon}^{(i)} + h'^{(i)}(\bar{\phi}^{(i)}) \odot \theta^{(i-1)^T} \bar{\varepsilon}^{(i-1)} \tag{2.20}$$

$$\frac{d\varepsilon^{(i)}}{dt} = \bar{\phi}^{(i)} - \theta^{(i)} h^{(i)}(\bar{\phi}^{(i+1)}) - \Sigma^{(i)} \bar{\varepsilon}^{(i)} \tag{2.21}$$

Notice that we have done away with the different update equations we had for ε_p and ε_u before by using internal state nodes $\bar{\phi}^{(i)}$ and variances $\Sigma^{(i)}$. Here, u would be denoted by $\bar{\phi}^{(0)}$ and Σ_u would be denoted by $\Sigma^{(0)}$. The variables v_p have all been replaced by $\theta h(\bar{\phi})$. Lastly, note that since we are now dealing with vectors of nodes instead of singular nodes, we use the Hadamard product $\odot$ in (2.20).

Bogacz derives the update to the interlayer connection weight strengths $\theta^{(i)}$ to be a tensor product of the error nodes $\bar{\varepsilon}^{(i)}$ on the lower layer with the state nodes $h(\bar{\phi}^{(i+1)})$ on the upper layer [4],

$$\frac{\partial F}{\partial \theta^{(i)}} = \bar{\varepsilon}^{(i)} h^{(i)}(\bar{\phi}^{(i+1)})^T \tag{2.22}$$

Note that the two terms in (2.22) correspond to the two sets of nodes $\bar{\varepsilon}^{(i)}$ and $h^{(i)}(\bar{\phi}^{(i+1)})$ that are presynaptic and postsynaptic neurons for the synapse represented by the connection weight strength $\theta^{(i)}$. Thus, it is a Hebbian update rule, preserving our biological plausibility requirement to have local synaptic plasticity.

Fig. 2.3 shows a Predictive Coding network with multiple neurons per layer that facilitates equations (2.20), (2.21), and (2.22).

So, why do these equations matter? How do they represent effective neural learning with respect to a given statistical learning task?

Bogacz explains this question by referring to Kullback-Leibler Divergence [31], which is more commonly known as KL Divergence. Kullback-Leibler Divergence is a measure of how one probability distribution $q(v)$ is different from a second probability distribution

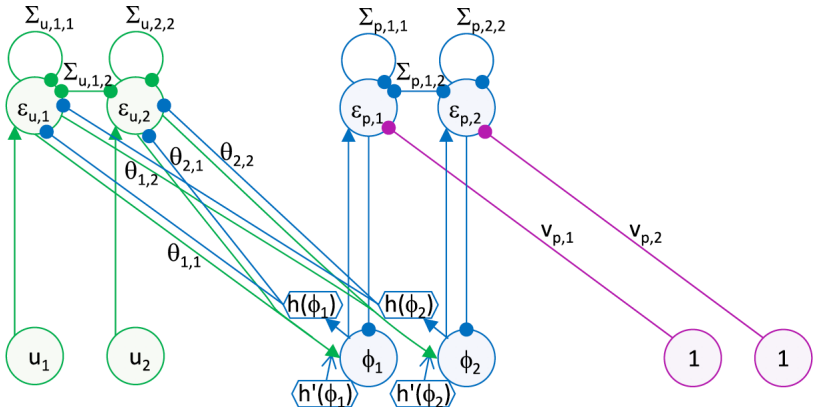


Figure 2.3: Predictive Coding neural network representing a model inferring 2 features from 2 sensory stimuli, taken from [4]. Green nodes represent the first layer in the hierarchy, blue nodes represent the second layer, and purple nodes represent the third layer.

$p(v|u)$ [32, 30, 4]. To simplify the following analysis, Bogacz assumes that $q(v)$ is a delta distribution which is ∞ at $v = \phi$ and 0 everywhere else. Now, the formula for KL divergence is

$$KL(q(v), p(v|u)) = \int q(v) \ln \frac{q(v)}{p(v|u)} dv \tag{2.23}$$

While any pair of distributions $q(x)$ and $p(x)$ could have been chosen for the KL Divergence formula, Bogacz chooses $q(v)$ and $p(v|u)$ since we are interested in estimating v from observing u. Substituting the definition of conditional probability, $p(v|u) = \frac{p(u,v)}{p(u)}$ into equation (2.23) and following through with the derivation [4] gives

$$KL(q(v), p(v|u)) = \int q(v) \ln \frac{q(v)}{p(u,v)} dv + \ln p(u) \tag{2.24}$$

The integral in (2.24) is actually Friston's concept of Free Energy, and Bogacz denotes its negative by F. Recall that F was defined as $\ln p(\phi) + \ln p(u|\phi)$. Under certain assumptions,

the negative Free Energy in (2.24) is equivalent to this F [4]. We write F now as the negative of the integral in (2.24).

$$F = \int q(v) \ln \frac{p(u,v)}{q(v)} dv \tag{2.25}$$

$$KL(q(v), p(v|u)) = -F + \ln p(u) \tag{2.26}$$

Recall that we assumed $q(v)$ is a delta distribution. This allows us to write F [4] as

$$\begin{aligned} F &= \int q(v) \ln \frac{p(u,v)}{q(v)} dv \\ &= \int q(v) \ln p(u,v) dv - \int q(v) \ln q(v) dv \\ &= \ln p(u,\phi) + C_1 \end{aligned}$$

The delta function with centre ϕ assumption implies that the integral of $q(v)$ multiplied by any function $h(x)$ is equal to $h(\phi)$, which, in this case, is $\ln p(u,\phi)$ [4]. The second integral does not depend on ϕ and so cancels out when we compute the derivative of F with respect to ϕ, so we denote it by a constant C_1 [4].

$$F = \ln p(u,\phi) + C_1 \tag{2.27}$$

By using the fact that $p(u, \phi) = p(\phi)p(u|\phi)$ and ignoring C_1 [4], we can then derive

$$\begin{aligned} F &= \ln p(u, \phi) \\ &= \ln p(\phi)p(u|\phi) \\ &= \ln p(\phi) + \ln p(u|\phi), \end{aligned}$$

which is exactly the same as (2.5).

Hence, the process of minimizing Free Energy, which is equivalent to maximizing F, lets us find the approximate delta distribution $q(v)$ [4]. Finding $q(v)$ allows us to retrieve ϕ, which is the reason for the whole set of equations (2.20), (2.21), and (2.22) behind our Predictive Coding network [4].

Another way to think about the network's inference process is by noting that, if we maximize $p(u)$, we are are finding a set of states in our neural network such that sensory observations are least surprising [4]. This follows from the fact that u is the noisy estimate of light intensity observed by our organism. Rearranging (2.26) as

$$\ln p(u) = F + KL(q(v), p(v|u)) \tag{2.28}$$

tells us that, since KL divergence is non-negative, F is a lower bound on $p(u)$, so by maximizing F, we maximize the lower bound on $p(u)$ [4]. Bogacz notes, however, that an organism actually wishes to maximize the average of $p(u)$ across multiple trials, or multiple attempts to find food, and so the parameters of its Predictive Coding network, which is presumably its brain, are only modified a little bit for each trial [4]. This is similar to

applying weight updates with a learning rate when training artificial neural networks with backpropagation.

2.2 Approximation of Backpropagation

Predictive Coding with the Free Energy Principle using Bogacz's equations allows us to design a network with a learning algorithm that approximates backpropagation. Whittington and Bogacz observe that, under certain conditions, if the equations used in backpropagation are written in a certain way and compared to the Predictive Coding equations at equilibrium, they are equivalent [68]. Now, we review the backpropagation equations and rewrite them.

Note that $\frac{\partial C}{\partial z_i}$ from (1.5) simplifies to $\bar{y} - \bar{t}$ for certain loss and activation functions [68]. Recall that $\bar{y}$ is the final output of the neural network and $\bar{t}$ is the target output. One such combination that allows for this simplification is logistic activation with cross entropy loss, which is commonly used for classification tasks. Another common combination is softmax activation with categorical cross entropy loss. Our assumption is therefore that a loss-activation function combination is used such that the derivative of the loss with respect to the current $\bar{z}$ entering the top layer is equal to $\bar{y} - \bar{t}$.

As we apply chain rule to get the derivative of the loss with respect to the hidden layers of the network, a simple pattern emerges. For each layer with entering current $\bar{z}^{(l)}$, if we have computed $\frac{\partial C}{\partial \bar{z}^{(l)}}$, then $\frac{\partial C}{\partial \bar{z}^{(l-1)}}$ is described in (1.6), and can be simplified to (1.8). For a neural network of n layers, with the input at layer 0, we can write each term $\frac{\partial C}{\partial \bar{z}^{(l)}}$ of

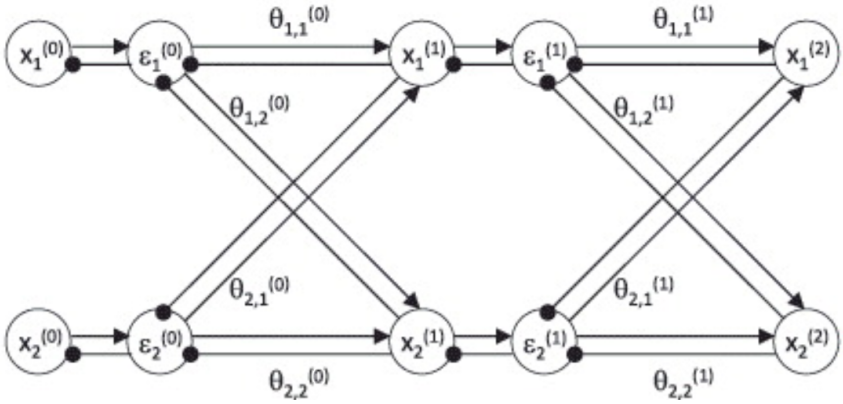


Figure 2.4: A Predictive Coding network taken from [68]. During training, the inputs $\bar{u}$ would be fixed to the two nodes on the right, $x^{(2)}$, while the outputs $\bar{t}$ would be fixed to the two nodes on the left, $x^{(0)}$. Arrows denote excitatory connections while lines ending with circles denote inhibitory connections. Connections without θ labels have weights fixed to 1.0.

backpropagation as follows.

$$\frac{\partial C}{\partial \bar{z}^{(l)}} = \begin{cases} \bar{y} - \bar{t} & l = n-1 \\ f'^{(l)}(\bar{x}^{(l-1)}) \odot W^{(l-1)^T} \frac{\partial C}{\partial \bar{z}^{(l+1)}} & 0 < l < n-1 \end{cases} \tag{2.29}$$

Now, we consider how our Predictive Coding model would learn. Since we are doing supervised learning, we have a dataset of inputs $\bar{u}$ and targets $\bar{t}$. Unlike what we had before, we engineer our training procedure such that we present the input $\bar{u}$ to layer $n-1$ and the target $\bar{t}$ to layer 0 of our Predictive Coding network. We will see the reasoning for this later.

Fig. 2.4 shows the network described in this paragraph and the next. Note that ϕ is shown as x in the figure, a switch of variable that Whittington and Bogacz make in their

paper on how Predictive Coding approximates backpropagation [68]. I will also be making this variable switch, so the following explanation will be based on the variables in Fig. 2.4.

When a stimulus has been presented to the network, its error nodes will acquire non-zero activity [4] and the values of the network's internal state nodes $\bar{x}^{(1)}$ will begin to change. During training, the value of the state nodes $\bar{x}^{(0)}$ will not change since they are fixed to $\bar{t}$. Similarly, the value of the state nodes $\bar{x}^{(2)}$ should not change since they are fixed to $\bar{u}$, though different implementations of holding the input to the network can exist.

At equilibrium, the error nodes $\bar{\varepsilon}^{(0)}$ on layer 0 will encode the difference between the target output and the output of the network. Let the output of the network be $\bar{\mu}$, and it is equal to the values of the internal state nodes of the previous layer $\bar{x}^{(1)}$ transformed by an activation function $h^{(0)}$ and multiplied by the connection weight strengths $\theta^{(0)}$.

$$\bar{\mu} = \theta^{(0)} h^{(0)}(\bar{x}^{(1)}) \tag{2.30}$$

Hence, the error nodes $\bar{\varepsilon}^{(0)}$ should equal $\bar{t} - \bar{\mu}$ at equilibrium. For every other layer, at equilibrium, the change in each error and state neuron should equal zero. Letting $\phi = x$ as before, we can set the left hand side in (2.20) to 0 and rearrange the equation to get

$$\bar{\varepsilon}^{(i)} = h'^{(i-1)}(\bar{x}^{(i)}) \odot \theta^{(i-1)^T} \bar{\varepsilon}^{(i-1)} \tag{2.31}$$

Unless the network has been trained to associate the input and the output, the error nodes will not converge to zero [4, 68] at equilibrium. Now we can write the value of each error

node at equilibrium as

$$\bar{\varepsilon}^{(i)} = \begin{cases} \bar{t} - \bar{\mu} & i = 0 \\ h'^{(i-1)}(\bar{x}^{(i)}) \odot \theta^{(i-1)^T} \bar{\varepsilon}^{(i-1)} & 0 < i < n-1 \end{cases} \tag{2.32}$$

Compare (2.29) and (2.32). They have essentially the same mathematical terms except with the layers reversed and with the target and network output switched in the difference expression. To account for the switch, the weight update $\frac{\partial F}{\partial \theta^{(i)}}$ is added to $\theta^{(i)}$ rather than subtracted as in backpropagation. This also makes sense under the Predictive Coding objective of maximizing F, the negative Free Energy, as opposed to backpropagation's objective of minimizing the loss C.

Now the question is whether or not those terms are actually driven by the Predictive Coding equations to equal the backprop terms at equilibrium in practice. Whittington and Bogacz investigate this question in their paper and find that the terms in their Predictive Coding network actually approximate backpropagation terms under some sets of circumstances [68]. Since the update to weight $\theta^{(i)}$ in the Predictive Coding network is the tensor product $\bar{\varepsilon}^{(i)} h^{(i)}(\bar{x}^{(i+1)})^T$, if the Predictive Coding terms equal the backprop terms, then the update will be the same as that used by backpropagation in (1.9).

This allows us to derive a similar learning algorithm for Predictive Coding networks as algorithm 1 [68]. Again, notice we switch the layers where we fix the inputs and targets:

Algorithm 2: Predictive Coding During Learning [68]

Initialize weights;

for *each epoch* **do**

  for *each batch u,t in dataset* **do**

    $\phi^{(n-1)} = \text{u}$;

    $\phi^{(0)} = \text{t}$;

    Inference: Equations (2.20), (2.21) until convergence

    Update weights: Equation (2.22)

  end

end

Observe that the learning algorithm involves fixing $\bar{u}$ and $\bar{t}$ to the network. This in effect clamps the network, preventing it from converging to an equilibrium where all the error nodes equal 0. If either $\bar{u}$ or $\bar{t}$ are removed, then there is no clamping, and the error nodes should all converge to 0. Bogacz shows that the equilibrium of the model is stable [4], and so running the Predictive Coding equations should always allow the network to converge to the equilibrium where all error nodes equal 0.

Consider also what happens when we fix only $\bar{u}$ or $\bar{t}$ to the network. Since an equilibrium to the Predictive Coding equations always exists, fixing either $\bar{u}$ or $\bar{t}$ to the network will cause all the internal state nodes to converge to some final value. At the other end of the network, $\bar{x}^{(0)}$ or $\bar{x}^{(n-1)}$, a result is produced. Usually, $\bar{x}^{(0)}$ is a class while $\bar{x}^{(n-1)}$ is an input such as an image.

Whittington and Bogacz have already shown that the Predictive Coding network is capable of learning image classification [68]: given an image at $\bar{x}^{(n-1)}$, the network predicts

the correct class at $\bar{x}^{(0)}$. Furthermore, since their model is the first conception of Predictive Coding that works with supervised learning, it is called **Supervised Predictive Coding**. But can their network perform the other task: generating good quality images at $\bar{x}^{(n-1)}$ when presented a class in $\bar{x}^{(0)}$?

We call generating images from classes the network's *generative* mode of operation [66], and the other mode in which it predicts the class from the image the *discriminative* mode of operation. In the following chapters, I will show the benefit of decay to both modes. Since Supervised Predictive Coding is already good at being discriminative, the main benefit of decay will be to make it generative.

In the next chapter, I will explain how decay can be introduced in two forms, *weight decay* and *activity decay* [66], and I will present the benefits that weight decay brings to the discriminative mode. In the chapter after that, I will show the pivotal role that both types of decay play in solving the problem of generating inputs from class vectors.

Chapter 3

Predictive Coding With Decay

What possible improvements can be made to Supervised Predictive Coding? Before we get to that, I will first discuss my implementation of Whittington and Bogacz's network.

Since a Predictive Coding neural network is a system of differential equations, it requires initial values in order to be numerically simulated. My convention is to set all the state and error nodes to zero at initialization.

When values such as input images or output classes are fixed to the state nodes at the top or bottom layers of the network, the network's internal state and error nodes begin to change. I simulate the differential equations that describe these changes with Forward Euler stepping until the equilibrium is reached.

Given a network of n layers, zero-indexed, the differential equations that govern the

deep layers of the network, which are layers i such that $1 \leq i \leq n-2$, are

$$\tau \frac{d\varepsilon^{(i)}}{dt} = x^{(i)} - M^{(i)}\sigma(x^{(i+1)}) - \nu^{(i)}\varepsilon^{(i)} \tag{3.1}$$

$$\tau \frac{dx^{(i)}}{dt} = \alpha^{(i)} W^{(i-1)}\varepsilon^{(i-1)} \odot \sigma'(x^{(i)}) - \beta^{(i)}\varepsilon^{(i)} \tag{3.2}$$

Notice that instead of using one weight matrix $\theta^{(i)}$ for forward and backward computations between layer (i) and layer $(i+1)$, I use matrix $M^{(i)}$ for the mapping from layer $(i+1)$ to layer (i) and matrix $W^{(i-1)}$ for the mapping from layer $(i-1)$ to layer (i). This resolves the problem of having symmetric weights, and using separate weights M and W still allows the network to achieve excellent performance. Similar to weight initialization in feedback alignment [37], I use random initialization of M and W.

For the other variables, $x^{(i)}$ denotes the vector of state nodes of layer i, $\varepsilon^{(i)}$ is the corresponding vector of error nodes for layer i, σ is an activation function such as tanh on $x^{(i)}$, $\nu^{(i)}$ is a scalar variance parameter that I always set to 1 to simplify my experiments, the $\odot$ operator represents the Hadamard product, and τ is a time constant.

The variables $\alpha^{(i)}$ and $\beta^{(i)}$ are binary valued gatekeepers used to simplify the computational aspect of the network switching between modes of operation. The variable $\alpha^{(i)}$ controls values being projected up to $x^{(i)}$ from lower layers, while $\beta^{(i)}$ controls values sent down to $x^{(i)}$ from error nodes $\varepsilon^{(i)}$ on the same layer. During learning, when the network is clamped, $\alpha^{(i)} = 1$ and $\beta^{(i)} = 1$ to allow state nodes $x^{(i)}$ to receive values from above and below for all i. The other two modes of operation are discussed later.

The equations for the input and output layers of the network are slightly different.

Exceptions are that equation (3.2) applies to the state nodes of layer $(n-1)$ and equation (3.1) applies to the error nodes of layer (0). It is the state nodes on layer (0) of the network and the error nodes on layer $(n-1)$ of the network that differ for reasons specified below.

The output of the network is the predicted class of the input and can be retrieved from the state nodes at layer (0) of the network at equilibrium. For the update to state nodes $x^{(0)}$, we simply take the negative of the corresponding error nodes $\varepsilon^{(0)}$ on that layer since there is no further layer below,

$$\tau \frac{dx^{(0)}}{dt} = -\beta^{(0)}\varepsilon^{(0)} \tag{3.3}$$

The input to the network is held in a vector termed *stim*, which stands for *stimulus*. For image classification, you can think of *stim* as the physical photons hitting the retina cells, while the cells starting from the retina correspond to the nodes of layer $(n-1)$.

Importantly, *stim* and the nodes of layer $(n-1)$ have the same dimensions. The update equation for the error nodes of layer $(n-1)$ is

$$\tau \frac{d\varepsilon^{(n-1)}}{dt} = x^{(n-1)} - stim - \nu^{(n-1)}\varepsilon^{(n-1)} \tag{3.4}$$

Now, we can discuss the settings of α and β in the network's other modes of operation. In discriminative mode, all α and β are set the same as the clamped mode except that $\alpha^{(n-1)} = 0$ in order that the state nodes $x^{(n-1)}$ exclusively receive input from *stim*. This is so that $x^{(n-1)}$ converges to the values of *stim* and so at equilibrium the output of the network is achieved from the correct setting of the input to the network. Note that there

is no need to set $\alpha^{(0)} = 0$ because there is no layer projecting values up from below $x^{(0)}$.

In generative mode, all α and β are set to the same as the clamped mode, but now $\beta^{(0)} = 0$ to prevent the state nodes $x^{(0)}$ that have been set to the values of class labels from changing, $\beta^{(n-1)} = 0$ to prevent the state nodes $x^{(n-1)}$ from receiving information from the error nodes that they are connected to that normally convey values from the now zero-valued *stim* container, and $\alpha^{(n-1)} = 1$ to allow state nodes $x^{(n-1)}$ to change to the inputs generated by the classes.

During learning, which is the network's clamped mode, I update the weights $M^{(i)}$ and $W^{(i)}$ in each time step using the gradients in (2.22). With asymmetric weights, the updates to $M^{(i)}$ and $W^{(i)}$ are transposes of each other,

$$\gamma \frac{dM^{(i)}}{dt} = \varepsilon^{(i-1)} \otimes \sigma(x^{(i)}) \tag{3.5}$$

$$\gamma \frac{dW^{(i)}}{dt} = \sigma(x^{(i)}) \otimes \varepsilon^{(i-1)} \tag{3.6}$$

The variable γ is a time constant similar to τ, but not necessarily equal since the synaptic changes in biological neural networks likely happen at a different rate than that of the changes to activities of nodes. It follows that my Predictive Coding training regime differs from that presented in algorithm 2 because it performs a weight update after every time step, rather than only after the state and error nodes have converged to equilibrium.

Fig. 3.1 summarizes the variables in my network, showing how they all connect together in a hierarchy of layers.

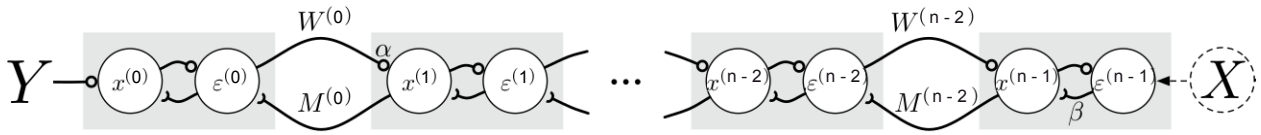


Figure 3.1: End to end schematic of my neural network. Notice how during learning, the class labels, Y, are fixed to the state nodes at layer 0, while the inputs, X, are placed in a container *stim* (dashed line circle) and "fed" into the error nodes on layer $(n-1)$. An example α is shown gating values fed up to state nodes and a sample β is shown gating values fed down to state nodes.

3.1 Decay in the Free Energy Objective

Recall that the objective of the Predictive Coding network is to maximize F, which can be defined as the sum of the negative of all the error nodes squared.

$$F = -\sum_{i=0}^{n-1} \frac{1}{2} \|\varepsilon^{(i)}\|^2 \tag{3.7}$$

Under the Whittington and Bogacz conception of Predictive Coding, the main relation that exists between nodes of the networks at equilibrium is

$$\varepsilon^{(i)} = \frac{x^{(i)} - M^{(i)}\sigma(x^{(i+1)})}{\Sigma^{(i)}} \tag{3.8}$$

This relation follows from fixing state nodes and setting all the differential equations for the error nodes (i.e. (3.1)), to zero.

The error nodes comprise half of the nodes in the Predictive Coding network. The other half consists of state nodes, and the remaining parameters of the network are the interlayer connection weights. Observe that the network's objective function only penalizes

the error nodes. What happens if we penalize all the parameters of the network? We can introduce constants $\lambda_x > 0$, $\lambda_M > 0$, and $\lambda_W > 0$ for each of those parameters and write a new objective function,

$$F = -\sum_{i=0}^{n-1} \frac{1}{2}\|\varepsilon^{(i)}\|^2 - \sum_{i=0}^{n-1} \lambda_x \|x^{(i)}\|^2 - \sum_{i=0}^{n-2} \lambda_M \|M^{(i)}\|^2 - \sum_{i=0}^{n-2} \lambda_W \|W^{(i)}\|^2 \tag{3.9}$$

Taking the derivative of (3.9) with respect to each parameter yields a slightly different set of update equations for the state nodes and weights. The state nodes $x^{(i)}$ now have a **decay** term in their update, $-\lambda_x x^{(i)}$. This term $\lambda_x x^{(i)}$ is *activity decay*, a phenomena related to spike-frequency adaptation in which the neuronal firing rate decreases for a stimulus of constant intensity [45]. Similarly, the weights have a decay in their updates, $-\lambda_M M^{(i)}$ and $-\lambda_W W^{(i)}$.

$$\tau \frac{dx^{(i)}}{dt} = \alpha^{(i)} W^{(i-1)} \varepsilon^{(i-1)} \odot \sigma'(x^{(i)}) - \beta^{(i)} \varepsilon^{(i)} - \boxed{\lambda_x x^{(i)}} \tag{3.10}$$

$$\tau \frac{dx^{(0)}}{dt} = -\beta^{(0)} \varepsilon^{(0)} - \boxed{\lambda_x x^{(0)}} \tag{3.11}$$

$$\gamma \frac{dM^{(i)}}{dt} = \varepsilon^{(i)} \otimes \sigma(x^{(i+1)}) - \boxed{\lambda_M M^{(i)}} \tag{3.12}$$

$$\gamma \frac{dW^{(i)}}{dt} = \sigma(x^{(i+1)}) \otimes \varepsilon^{(i)} - \boxed{\lambda_W W^{(i)}} \tag{3.13}$$

What can be the effects of these decay terms? Weight decay [49] is a well-known regularization technique. In artificial neural networks, weight decay is usually formulated as a penalty for large weight and bias parameters in the loss function. Given a loss function $C(x, t; \theta)$ for inputs x, targets t, and weights and biases θ, a minimum-norm penalty for

parameters θ can be introduced into the loss,

$$C(x;\theta) = C(x,t;\theta) + \boxed{\lambda\|\theta\|^2} \tag{3.14}$$

This new cost function changes the backpropagation update to

$$\theta_i \leftarrow \theta_i - \alpha\frac{\partial C}{\partial \theta_i} - \boxed{2\lambda\theta_i} \tag{3.15}$$

The benefits of weight decay in artificial neural networks are numerous, including better generalization to unseen data points from test datasets [16] and even reducing classification error on the training dataset [29]. Can similar benefits be seen from adding weight and activity decay to Predictive Coding networks?

Weight decay has a clear effect on Predictive Coding training by altering the update to the weights, $\frac{dM^{(i)}}{dt}$ and $\frac{dW^{(i)}}{dt}$, when $\lambda_M > 0$ and $\lambda_W > 0$. We will see that weight decay confers significant benefits to Predictive Coding networks in this chapter and the next. But first, we will analyze the impact of activity decay on training.

3.2 Analysis of Activity Decay During Training

In this section, I will explain how activity decay has a detrimental effect on training. First, I will show that the error nodes $\varepsilon^{(i)}$ do not go to zero during training. We will use a proof by contradiction. Then, I will demonstrate how non-zero error nodes cause activity decay to negatively impact training.

Consider a 2-h-3 network, where the hidden layer contains h state nodes $x^{(1)}$ and h error nodes $\varepsilon^{(1)}$. The network uses linear activations σ and has an activity decay of λ_x on $x^{(1)}$. During training, inputs X and output classes Y are fixed to the network's 3-node layer and 2-node layer respectively. At the input layer, $\alpha = 0$ and $\beta = 1$. As a result, $x^{(2)}$ receives no input from the hidden layer, so it is only updated via its connecting error nodes $\varepsilon^{(2)}$. At the class layer, $\beta = 0$, so $x^{(0)}$ receives no update. For all other nodes, $\alpha = 1$ and $\beta = 1$.

The equations that govern the network during training are

$$\tau \frac{d\varepsilon^{(2)}}{dt} = x^{(2)} - X - \nu^{(2)}\varepsilon^{(2)} \tag{3.16}$$

$$\tau \frac{dx^{(2)}}{dt} = -\varepsilon^{(2)} \tag{3.17}$$

$$\tau \frac{d\varepsilon^{(1)}}{dt} = x^{(1)} - M^{(1)}\sigma(x^{(2)}) - \nu^{(1)}\varepsilon^{(1)} \tag{3.18}$$

$$\tau \frac{dx^{(1)}}{dt} = W^{(0)}\varepsilon^{(0)} \odot \sigma'(x^{(1)}) - \varepsilon^{(1)} - \lambda_x x^{(1)} \tag{3.19}$$

$$\tau \frac{d\varepsilon^{(0)}}{dt} = x^{(0)} - M^{(0)}\sigma(x^{(1)}) - \nu^{(0)}\varepsilon^{(0)} \tag{3.20}$$

$$\tau \frac{dx^{(0)}}{dt} = 0 \tag{3.21}$$

To start the proof by contradiction, we assume that all error nodes are 0 at the equilibrium during training. At the equilibrium, (3.16) equals 0 which implies $x^{(2)} = X$. Setting (3.18) to 0 and with the assumption that $\varepsilon^{(1)} = 0$ gives us

$$x^{(1)} = M^{(1)}\sigma(X) \tag{3.22}$$

We are interested in finding all the constraints on $x^{(1)}$. Since σ is linear, equation (3.22) gives exactly h constraints on $x^{(1)}$, so currently, $x^{(1)}$ is fully determined. But are there additional constraints on $x^{(1)}$? Observe that since the class Y is fixed to $x^{(0)}$, $x^{(0)} = Y$. Setting (3.20) to 0 and with the assumption that $\varepsilon^{(0)} = 0$ gives,

$$Y = M^{(0)}\sigma(x^{(1)}) \tag{3.23}$$

Since $M^{(0)}$ is a $2 \times h$ matrix, there are now two additional constraints on $x^{(1)}$. Given linear σ, the system is overdetermined with two extra constraints. There only exists a solution for $x^{(1)}$ if the constraints in (3.23) are consistent with the constraints in (3.22), but there is no reason to assume this is ever the case. But we also know that the network is stable [4], so an equilibrium solution for $x^{(1)}$ must exist. By contradiction, the assumption that all the error nodes $\varepsilon^{(i)}$ are 0 at the equilibrium during training must be false.

This result holds as long as the class layer has $j > 0$ nodes. In fact, the class layer always imposes j additional constraints on $x^{(1)}$ than the number of variables in $x^{(1)}$, so a linear network with at least one hidden layer will always be overdetermined. Training will therefore find an equilibrium solution for $x^{(1)}$ that compromises between all the constraints on $x^{(1)}$. This compromise solution will almost never make all the error nodes go to zero, though the errors can get very close to zero.

Assume that the network has been trained via the regular Predictive Coding training process without any decay. Then its errors will be zero on average at the equilibrium. Consider (3.19) at equilibrium. With the error nodes $\varepsilon^{(1)}$ and $\varepsilon^{(0)}$ being zero on average,

the largest term in (3.19) would be $\lambda_x x^{(1)}$. Enforcing equilibrium, we get in (3.19) that

$$\lambda_x x^{(1)} = 0 \tag{3.24}$$

The only solution for (3.24) is if $x^{(1)} = 0$. However, as long as Y is not all zeros, the network cannot map $x^{(1)}$ to Y if $x^{(1)}$ is all zeros, and so $x^{(1)}$ takes on at least some non-zero values.

Due to $\lambda_x > 0$, the state nodes $x^{(1)}$ are pushed towards zero, pulled away from their former equilibrium point. This pulling affects the error nodes by drawing them away from zero. These non-zero errors accumulate in weight matrices M and W, and can cause them to grow unstably. But the unbounded growth can be counter-balanced by weight decay, which during training will yield a unique, low-error equilibrium for the network, as well as unique weights [66].

This analysis provides us with two findings. First, using linear networks, the errors will almost never fully reach zero during training due to the overdetermined hidden layers. This result is not necessarily a deal breaker; Predictive Coding networks are still capable of learning input-output associations to a high degree of accuracy [68]. Second, activity decay is detrimental to learning, and only weight decay should be applied during training. In the next chapter, I will present the benefit of activity decay outside the training context.

Now that I have examined the effect of decay on training, we can next examine the effect of decay on the discriminative and generative modes. In the rest of this chapter, I will show that weight decay training improves the network's convergence to equilibrium in the discriminative mode. The equilibrium of the discriminative mode is where the network decides the class of the input, so it is important that the network can converge to

equilibrium in a reasonable amount of time. The effect of decay on the generative mode will be explored in the next chapter.

3.3 Stability Analysis of the Discriminative Mode

For simplicity, I restrict my analysis of the discriminative mode to the case of linear networks, where every activation function σ used in the network is linear. The analysis is far more difficult if activation functions are non-linear, as then multiple equilibria can exist. That analysis will be relegated to future work.

Dynamical systems analysis is a useful tool to analyze the stability of the network. Note that Bogacz proved that the network is always stable [4]. What I aim to do is show that, with a bad choice of hyperparameters, the network will take a long time to reach the equilibrium. Introducing decay can alleviate this problem even with bad hyperparameters.

3.3.1 Linear Stability Analysis of Dynamical Systems

Consider a system of differential equations of two variables, x and y.

$$\dot{x} = f(x, y) \tag{3.25}$$

$$\dot{y} = g(x, y) \tag{3.26}$$

Here, f and g can be any function. Importantly, note that at the equilibrium $(\bar{x}, \bar{y})$, $f(\bar{x}, \bar{y}) = g(\bar{x}, \bar{y}) = 0$ since the values of x and y no longer change. Now, consider adding

some small perturbation (X, Y) to $(\bar{x}, \bar{y})$ from their equilibrium points. Then we can use Taylor's theorem to create an approximation of $f(X + \bar{x}, Y + \bar{y})$,

$$\dot{x} = f(\bar{x}, \bar{y}) + \frac{1}{2}\begin{bmatrix} \frac{\partial f}{\partial x}(\bar{x}, \bar{y}) & \frac{\partial f}{\partial y}(\bar{x}, \bar{y}) \end{bmatrix}\begin{bmatrix} X \\ Y \end{bmatrix} + \mathcal{O}(X^2, Y^2) \tag{3.27}$$

$$\dot{y} = g(\bar{x}, \bar{y}) + \frac{1}{2}\begin{bmatrix} \frac{\partial g}{\partial x}(\bar{x}, \bar{y}) & \frac{\partial g}{\partial y}(\bar{x}, \bar{y}) \end{bmatrix}\begin{bmatrix} X \\ Y \end{bmatrix} + \mathcal{O}(X^2, Y^2) \tag{3.28}$$

Both $f(\bar{x}, \bar{y})$ and $g(\bar{x}, \bar{y})$ will equal 0. We can more succinctly write equations (3.27) and (3.28) in matrix form,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} \frac{\partial f}{\partial x}(\bar{x}, \bar{y}) & \frac{\partial f}{\partial y}(\bar{x}, \bar{y}) \\ \frac{\partial g}{\partial x}(\bar{x}, \bar{y}) & \frac{\partial g}{\partial y}(\bar{x}, \bar{y}) \end{bmatrix}\begin{bmatrix} X \\ Y \end{bmatrix} + \mathcal{O}(X^2, Y^2) \tag{3.29}$$

Equation (3.29) tells us that the dynamics of (x, y) are largely determined by the Jacobian, their first derivative matrix. The order $\mathcal{O}(X^2, Y^2)$ terms do not contribute significantly to the approximation as the perturbation from $(\bar{x}, \bar{y})$ gets smaller. Now, we can use matrix exponentiation to uncover how the Jacobian determines the stability of the system near its equilibrium.

Consider the following system, where a, b, c, d can take any value:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} \tag{3.30}$$

Denote the 2×2 matrix in (3.30) as A. Differential equations theory tells us that the solution to this system, which we denote as $\vec{x}(t)$ (the variable y is included in $\vec{x}(t)$), is ce^{At} for some constant c [41]. The matrix exponential function is defined in this manner: if A is an $n \times n$ diagonal matrix with r_1, r_2, ..., r_n down its main diagonal, then e^{At} is the diagonal matrix with $e^{r_1 t}$, $e^{r_2 t}$, ..., $e^{r_n t}$ down its main diagonal [41]. If A is not a diagonal matrix, then it can be diagonalized into the form PDP^{-1} where D is a diagonal matrix if P exists such that $A = PDP^{-1}$. The values of the diagonal of D will be the eigenvalues of A. Experimentally, for the Predictive Coding equations, the matrix is almost always diagonalizable.

Ignoring t as a scalar, we can write the Taylor expansion of $e^{PDP^{-1}}$ as

$$\begin{aligned} e^{PDP^{-1}} &= I + PDP^{-1} + \frac{1}{2!}[PDP^{-1}]^2 + \frac{1}{3!}[PDP^{-1}]^3 + \dots \\ &= I + PDP^{-1} + \frac{1}{2!}PD^2P^{-1} + \frac{1}{3!}PD^3P^{-1} + \dots \\ &= P\left[I + D + \frac{1}{2!}D^2 + \frac{1}{3!}D^3 + \dots\right]P^{-1} \\ &= Pe^{D}P^{-1} \end{aligned}$$

Let $r_1 = \alpha + i\beta$ and $r_2 = \alpha - i\beta$ be eigenvalues of (3.30). These two eigenvalues will be along the diagonal of the matrix D in $Pe^{Dt}P^{-1}$, derived above. The eigenvalues of D are mapped into the space of A via pre- and post-multiplication by P and P^{-1} respectively. Thus, the eigenvalues r_1 and r_2 determine the stability of (3.30) by affecting e^{At}.

The important value here is α, since $e^{(i\beta)t} = \sin \beta t + i \cos \beta t$, which is sinusoidal and so $e^{\alpha t}$ primarily determines whether the system is stable. If $\alpha > 0$, then the system is

unstable and will diverge from the equilibrium. If $\alpha < 0$, then the system is stable and will "spiral" towards the equilibrium. Even if the eigenvalues are not complex, the sign of α still determines the stability behaviour.

Since (3.29) takes a form similar to (3.30), putting our knowledge of the effect of α together with those two equations tells us that the stability of the Jacobian in (3.29) is determined by the eigenvalues of the Jacobian. Thus, to determine if a linear dynamical system is stable around its equilibrium, we will need the eigenvalues of the Jacobian at equilibrium. In the next section, I consider the Predictive Coding equations in a form similar to (3.30) and investigate their stability.

3.4 Predictive Coding Discriminative Dynamics

I will now show that the Predictive Coding equations with linear activations can be put in a form similar to (3.29) since the change to each variable is just a linear combination of every other variable. For this analysis, I consider a linear 2-3-4 network, which means a network with 2 state and 2 error nodes at layer 0, 3 state and 3 error nodes at layer 1, and 4 state and 4 error nodes at layer 2. In total, there are 18 variables: 9 state nodes and 9

error nodes. First, the equations for each variable individually are shown below.

$$\dot{x}_0^{(0)} = -\beta_0^{(0)} e_0^{(0)} \tag{3.31}$$

$$\dot{x}_1^{(0)} = -\beta_1^{(0)} e_1^{(0)} \tag{3.32}$$

$$\dot{x}_0^{(1)} = -\beta_0^{(1)} e_0^{(1)} + \alpha_0^{(1)} \sum_{i=0}^{1} e_i^{(0)} W_{i,0}^{(0)} f'\left(x_0^{(1)}\right) \tag{3.33}$$

$$\dot{x}_1^{(1)} = -\beta_1^{(1)} e_1^{(1)} + \alpha_1^{(1)} \sum_{i=0}^{1} e_i^{(0)} W_{i,1}^{(0)} f'\left(x_1^{(1)}\right) \tag{3.34}$$

$$\dot{x}_2^{(1)} = -\beta_2^{(1)} e_2^{(1)} + \alpha_2^{(1)} \sum_{i=0}^{1} e_i^{(0)} W_{i,2}^{(0)} f'\left(x_2^{(1)}\right) \tag{3.35}$$

$$\dot{x}_0^{(2)} = -\beta_0^{(2)} e_0^{(2)} + \alpha_0^{(2)} \sum_{i=0}^{2} e_i^{(1)} W_{i,0}^{(1)} f'\left(x_0^{(2)}\right) \tag{3.36}$$

$$\dot{x}_1^{(2)} = -\beta_1^{(2)} e_1^{(2)} + \alpha_1^{(2)} \sum_{i=0}^{2} e_i^{(1)} W_{i,1}^{(1)} f'\left(x_1^{(2)}\right) \tag{3.37}$$

$$\dot{x}_2^{(2)} = -\beta_2^{(2)} e_2^{(2)} + \alpha_2^{(2)} \sum_{i=0}^{2} e_i^{(1)} W_{i,2}^{(1)} f'\left(x_2^{(2)}\right) \tag{3.38}$$

$$\dot{x}_3^{(2)} = -\beta_3^{(2)} e_3^{(2)} + \alpha_3^{(2)} \sum_{i=0}^{2} e_i^{(1)} W_{i,3}^{(1)} f'\left(x_3^{(2)}\right) \tag{3.39}$$

$$\dot{e}_0^{(0)} = x_0^{(0)} - \sum_{i=0}^{2} f\left(x_i^{(1)}\right) M_{i,0}^{(0)} - e_0^{(0)} \tag{3.40}$$

$$\dot{e}_1^{(0)} = x_1^{(0)} - \sum_{i=0}^{2} f\left(x_i^{(1)}\right) M_{i,1}^{(0)} - e_1^{(0)} \tag{3.41}$$

$$\dot{e}_0^{(1)} = x_0^{(1)} - \sum_{i=0}^{3} f\left(x_i^{(2)}\right) M_{i,0}^{(1)} - e_0^{(1)} \tag{3.42}$$

$$\dot{e}_1^{(1)} = x_1^{(1)} - \sum_{i=0}^{3} f\left(x_i^{(2)}\right) M_{i,1}^{(1)} - e_1^{(1)} \tag{3.43}$$

$$\dot{e}_2^{(1)} = x_2^{(1)} - \sum_{i=0}^{3} f\left(x_i^{(2)}\right) M_{i,2}^{(1)} - e_2^{(1)} \tag{3.44}$$

$$\dot{e}_0^{(2)} = x_0^{(2)} - stim_0 - e_0^{(2)} \tag{3.45}$$

$$\dot{e}_1^{(2)} = x_1^{(2)} - stim_1 - e_1^{(2)} \tag{3.46}$$

$$\dot{e}_2^{(2)} = x_2^{(2)} - stim_2 - e_2^{(2)} \tag{3.47}$$

$$\dot{e}_3^{(2)} = x_3^{(2)} - stim_3 - e_3^{(2)} \tag{3.48}$$

Equations (3.31)-(3.48) can be written in **Sympy**, where the change with respect to time of all 18 variables is stored in an 18-vector, and the differential equations for each variable is stored in a corresponding 18-vector. Then we can compute the Jacobian, the 18×18 matrix of derivatives of each variable with respect to every other variable. The code for computing the Jacobian, using roughly similar indexing conventions as in equations (3.31)-(3.48), is shown in Fig. 3.2.

Since I am interested in analyzing the stability of the network near equilibrium, the values that will be substituted into each variable of the Sympy Matrix will be those of the PC network nodes at equilibrium given an input fixed into the *stim* container. The

```
DEs = sp.Matrix([-1.0*B0_0*e0_0,
            -1.0*B1_0*e1_0,
            -1.0*B0_1*e0_1 + A0_1*fp0_1*(e0_0*W0[0][0] + e1_0*W0[1][0]),
            -1.0*B1_1*e1_1 + A1_1*fp1_1*(e0_0*W0[0][1] + e1_0*W0[1][1]),
            -1.0*B2_1*e2_1 + A2_1*fp2_1*(e0_0*W0[0][2] + e1_0*W0[1][2]),
            -1.0*B0_2*e0_2 + A0_2*fp0_2*(e0_1*W1[0][0] + e1_1*W1[1][0] + e2_1*W1[2][0]),
            -1.0*B1_2*e1_2 + A1_2*fp1_2*(e0_1*W1[0][1] + e1_1*W1[1][1] + e2_1*W1[2][1]),
            -1.0*B2_2*e2_2 + A2_2*fp2_2*(e0_1*W1[0][2] + e1_1*W1[1][2] + e2_1*W1[2][2]),
            -1.0*B3_2*e3_2 + A3_2*fp3_2*(e0_1*W1[0][3] + e1_1*W1[1][3] + e2_1*W1[2][3]),
            x0_0 - (f0_1*M0[0][0] + f1_1*M0[1][0] + f2_1*M0[2][0]) - e0_0,
            x1_0 - (f0_1*M0[0][1] + f1_1*M0[1][1] + f2_1*M0[2][1]) - e1_0,
            x0_1 - (f0_2*M1[0][0] + f1_2*M1[1][0] + f2_2*M1[2][0] + f3_2*M1[3][0]) - e0_1,
            x1_1 - (f0_2*M1[0][1] + f1_2*M1[1][1] + f2_2*M1[2][1] + f3_2*M1[3][1]) - e1_1,
            x2_1 - (f0_2*M1[0][2] + f1_2*M1[1][2] + f2_2*M1[2][2] + f3_2*M1[3][2]) - e2_1,
            x0_2 - stim0 - e0_2,
            x1_2 - stim1 - e1_2,
            x2_2 - stim2 - e2_2,
            x3_2 - stim3 - e3_2,])

variables = sp.Matrix([x0_0, x1_0, x0_1, x1_1, x2_1, x0_2, x1_2, x2_2, x3_2,
                           e0_0, e1_0, e0_1, e1_1, e2_1, e0_2, e1_2, e2_2, e3_2])

Jacobian = DEs.jacobian(variables)
```

Figure 3.2: Equations (3.31)-(3.48) in a Sympy (sp) Matrix, ready to have values substituted into each variable. Note that the values in the weight matrices are retrieved through array indexing, and this is done because the entire weight matrix is stored as a single variable such as W0, the weights that map error nodes from layer 0 to layer 1.

experiments in the next section will show that the capacity of the network to converge to equilibrium is dependent on the eigenvalues of the Jacobian of the above matrix, and that these eigenvalues are greatly improved through the introduction of decay into the network.

3.4.1 Stability of Predictive Coding Trained Weights

Before the main experiment, we do a preliminary experiment to see how the network converges with a bad hyperparameter choice.

I start by creating a dataset with four-dimensional inputs and two-dimensional outputs. The inputs are created by sampling two random four-dimensional vectors where each value is drawn from a uniform distribution on $[-0.5, 0.5)$ and multiplying the result by 3. The classes are the identity 2×2 matrix, so the first input vector corresponds to the $[1, 0]$ one-hot class and the second input vector corresponds to the $[0, 1]$ one-hot class. Additional training points are created by sampling noise drawn from a normal distribution with 0 mean and variance 0.2 until 300 training points have been created.

Next, I train a Predictive Coding network for 10 epochs without weight decay or activity decay. For each input batch, which consisted of 30 data points, I clamped the input sample and output class to the network and ran the Predictive Coding equations for 5 seconds of time (recall that τ and γ are all in units of seconds) using $\tau = 0.2$, $\gamma = 0.08$, and a time step of 0.001. The bad hyperparameter choice here is $\tau = 0.2$. From prior experiments, I know that this value is too large to allow for fast convergence times on this dataset.

Once the network is trained to classify the training dataset with perfect accuracy, I generate another input vector which uses a linear combination of the rows of the two input

vectors with some normal-sampled noise added. This ambiguous input is meant to be a data point somewhere between the two input vectors, and so it is very likely to be a point that the network has not been trained on before. It also does not have a corresponding class value. What I am interested in is seeing how the network converges on such an ambiguous input.

While this input point is fixed to the *stim* container, I simulate the Predictive Coding equations in the **discriminative mode** with $\tau = 0.2$ for 10 seconds and time steps of 0.001 and observe the norm of the values of each state and error node in each time step. No decay is used in the state node updates.

What I should expect to see is that the norm of the values of the error nodes go to zero as F is maximized while the norm of the values of the state nodes become stable at some constant. The results are displayed in Fig. 3.3. Note that the norm of $\varepsilon^{(2)}$ is always zero after each time step since these nodes always fully transmit information from the *stim* container to the state nodes $x^{(2)}$ in the discriminative mode where $\alpha^{(n-1)} = 0$, so the graph of $\varepsilon^{(2)}$ is not shown (it would just be blank).

Even after ten thousand time steps, the error nodes $\varepsilon^{(0)}$ and $\varepsilon^{(1)}$ fail to reach zero, and the state nodes $x^{(0)}$ and $x^{(1)}$ have not yet plateaued to a constant value. The state nodes $x^{(2)}$ have plateaued since they just converge quickly to the values at the *stim* container (i.e. the input vector fixed to the network). The sum of the norms of all the error nodes is 0.0428. This suggests the network is not close to the equilibrium.

Fortunately, we already know what the equilibrium values of the network will be. Why is that?

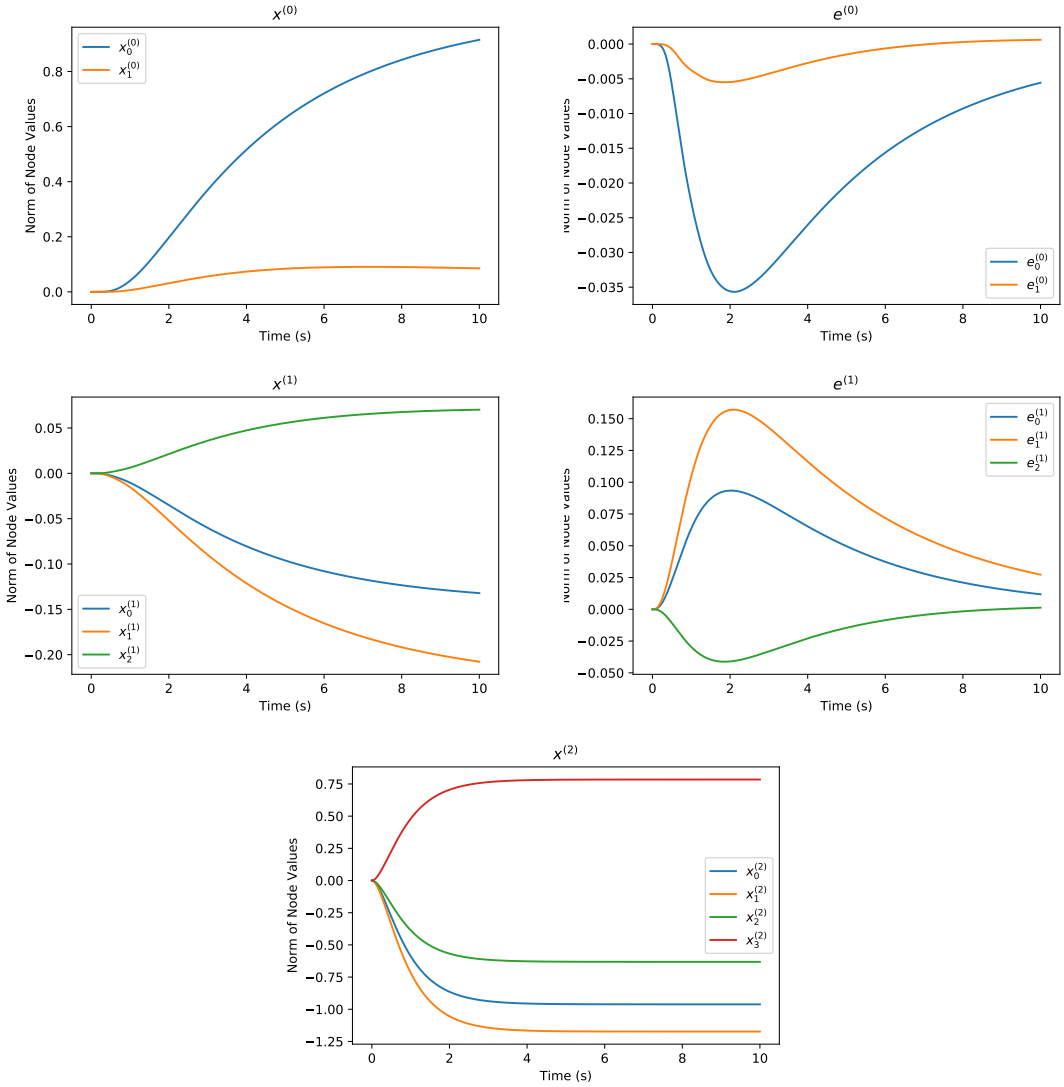


Figure 3.3: Graphs of the norms of each layer's state and error nodes with respect to time during discriminative mode of operation. Ten thousand time steps were taken in total.

Consider the relation between state and error nodes in (3.8). If we want all the error nodes to be zero, we simply set all state nodes $x^{(i)}$ to $M^{(i)}\sigma(x^{(i+1)})$. This can be done iteratively starting from the state nodes $x^{(2)}$ for this network. The benefit of knowing the equilibrium is that we can directly observe the eigenvalues of the Jacobian at the equilibrium.

Using the matrix outlined in Fig. 3.2, I can compute the Jacobian at the equilibrium with the values of the network's parameters at equilibrium, and then numerically solve for the eigenvalues of the Jacobian. The results are shown in table 3.1.

Table 3.1: Eigenvalues of the Jacobian of a non-decay trained Predictive Coding network at equilibrium.

Eigenvalue	**Real Component**	**Imaginary Component**
1	-0.5000	4.855
2	-0.5000	-4.855
3	-0.5000	3.763
4	-0.5000	-3.763
5	-0.0412	0
6	-0.0861	0
7	-0.9139	0
8	-0.9588	0
9	-0.5000	0.7987
10	-0.5000	-0.7987
11	-0.5000	0.9102
12	-0.5000	-0.9102
13	-0.5000	0.8494
14	-0.5000	-0.8494
15	-0.5000	0.8707
16	-0.5000	-0.8707
17	-0.5000	0.8660
18	-0.5000	-0.8660

Notice that the real components of eigenvalues 5 and 6 are -0.0412 and -0.0861 respectively, which are relatively close to 0. For some constant $c > 0$, the function $ce^{-0.0412t}$ goes to 0 very slowly as t increases. So, despite being a stable system, the Predictive Coding network will be slow at converging to equilibrium. This is what we observe in Fig. 3.3 with the errors going towards zero very slowly in ten thousand time steps.

Importantly, the network is a trained Predictive Coding network, which means the weights have learned the 4-to-2 classification task with a hundred percent accuracy. If the weight matrices have a high norm, they may contribute to the existence of extreme-valued eigenvalues in the Jacobian at equilibrium such as those that are close to 0.

What if we now train the network with decay? We know that decay helps to regularize the weights in artificial neural networks, punishing weights with extreme values and pushing the optimization landscape towards weights with less extreme values. Adding weight decay should result in better eigenvalues that are further away from zero.

I hypothesize that adding decay to Predictive Coding networks will improve the eigenvalues of the Jacobian of the network equations at equilibrium such that, with increasing decay, the maximum eigenvalue decreases, approaching -0.5.

Experimental Setup: I trained ten networks using the same hyperparameters outlined in the initial experiment of this section, except for the decay hyperparameters. An increasing *weight decay* is applied starting from $\lambda_M = \lambda_W = 0.003$ up to $\lambda_M = \lambda_W = 0.03$ at steps of 0.003. Activity decay is fixed at $\lambda_x = 0.0$ for all networks, since for linear networks, I avoid applying activity decay during training.

After each network is trained, I create an ambiguous input which I fix to the network's

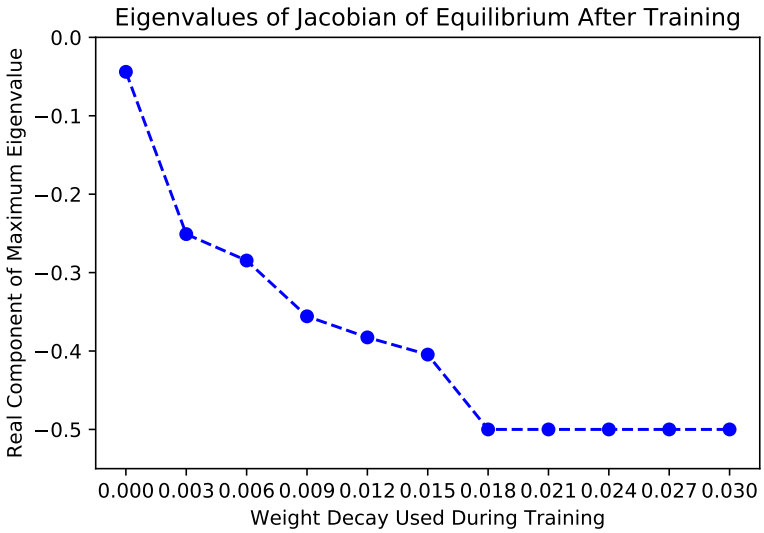


Figure 3.4: Real components of the eigenvalues of the Jacobian of the Predictive Coding network at equilibrium in discriminative mode with an input fixed after training with specified weight decay.

stim container and compute the eigenvalues of the Jacobian of the network at the equilibrium point in **discriminative mode**. The equilibrium is found by setting all the state nodes to the values they would be at when the error nodes are zero with the ambiguous input fixed. The eigenvalue with the highest real component is recorded.

Next, starting from every state and error node at zero, I simulate the network in **discriminative mode** with the ambiguous input fixed and see if it reaches the equilibrium computed earlier. To verify that it reaches the equilibrium, I record the sum of the norms of all the network's error nodes after 10 simulation seconds with a time step of 0.001 seconds. If the sum is around 0.001 or less, then the network is reasonably close to equilibrium.

Results: Fig. 3.4 displays the maximum eigenvalue of the Jacobian of the equilibrium of the PC equations with an input fixed in the discriminative mode of operation after

training. When there is no weight decay during training ($\lambda_M = \lambda_W = 0.0$), the real component of the maximum eigenvalue is close to zero, at -0.0439. As the value of the weight decay used during training increases, the maximum eigenvalue decreases. At a weight decay of $\lambda_M = \lambda_W = 0.018$ and above, the maximum eigenvalue stays at -0.5, which is ideal for convergence to equilibrium.

Table 3.2: Sums of the norms of the error nodes after ten simulation seconds in discriminative mode after weight decay training.

Weight Decay Used During Training	**Norms of Error Nodes**
0.000	0.0429
0.003	0.0015
0.006	0.0010
0.009	0.0006
0.012	0.0005
0.015	0.0005
0.018	0.0004
0.021	0.0004
0.024	0.0004
0.027	0.0004
0.030	0.0003

These results are consistent with my hypothesis that increasing weight decay will improve the eigenvalues of the Jacobian at equilibrium until all the real components of the eigenvalues are at -0.5. But does this result in better convergence of the network's nodes to equilibrium?

Table 3.2 displays the sums of the norms of the error nodes of the Predictive Coding network after it has been simulated in the discriminative mode of operation with an input fixed to its *stim* container for each weight decay value used during training. A lower sum

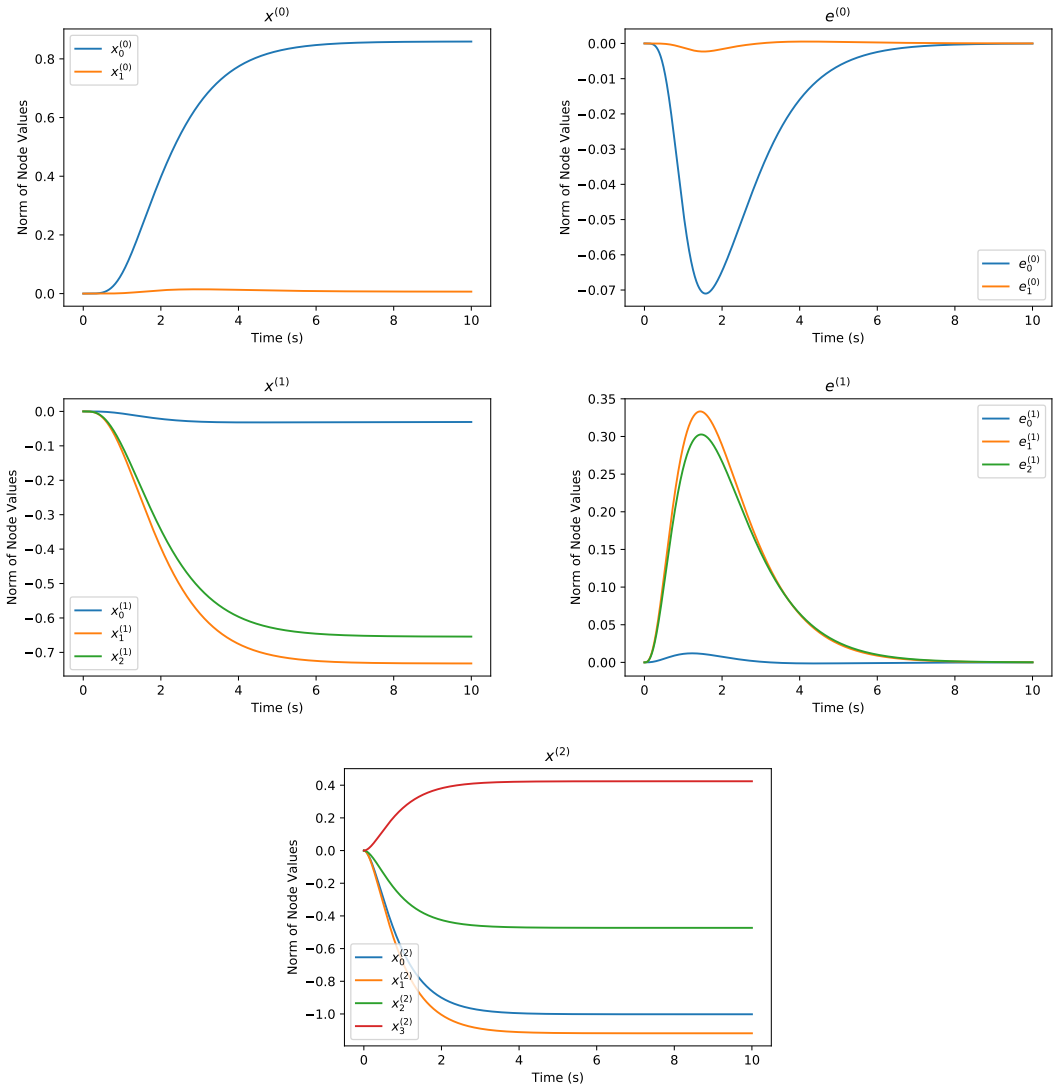


Figure 3.5: Graphs of the norms of each layer's state and error nodes with respect to time during discriminative mode of operation after training with weight decay $\lambda_M = \lambda_W = 0.03$. Ten thousand time steps were taken in total.

is better since it implies that the error nodes have mostly gone to zero and so the network is close to equilibrium.

As soon as a weight decay value of 0.003 is used, the sum decreases by more than an order of a magnitude. Weight decay training significantly regularizes the parameters of the network, making it much easier for the error nodes to converge to zero when the network is classifying an input.

The graphs of the norms of each of the network node values by time step are shown in Fig. 3.5, this time using a network trained with a weight decay of 0.03. Unlike before, all the error nodes now converge relatively close to 0, and all the state nodes succeed in reaching a constant value.

These results fortify the theoretical relation discussed earlier between the eigenvalues of the Jacobian of the network equations at equilibrium and the capability of the network to converge to the equilibrium. With eigenvalues whose real components are all at -0.5, the network converges smoothly, and weight decay training is the catalyst for improving all the eigenvalues.

On this simple dataset, the problem of convergence to equilibrium can certainly be mitigated in a simpler way by choosing a better set of hyperparameters, such as a smaller time constant τ. But this fix may not work on more complex, higher-dimensional datasets, or on different network topologies such as convolutional layers [35] applied to a Predictive Coding architecture. For those datasets and networks, weight decay may be necessary to facilitate convergence to equilibrium.

The goal of this experiment is to show that weight decay has a regularizing effect on

the weights of Predictive Coding networks, just like it does in artificial neural networks. We will see in the next chapter that this regularization entails a much more important benefit for the generative mode of operation of Predictive Coding.

3.4.2 Stability of Backpropagation Trained Weights

Before closing this chapter, I want to consider what happens if we train an artificial neural network with backpropagation, and then import the trained weights into a Predictive Coding neural network. Why is this something worth considering? Since Supervised Predictive Coding approximates backpropagation, a trained Predictive Coding network should, under certain conditions, have theoretically similar weights as an artificial neural network trained with backpropagation on the same data. However, *similar* does not imply *exact*. If there are differences, how would they manifest?

Using the same dataset as in the previous section, I trained an artificial neural network with linear activations and mean squared error loss using backpropagation with a learning rate of 0.01 and no decay for 500 epochs with a batch size of 30. The neural network successfully learns the classification task and achieves 100% accuracy. Then, I imported those network weights into a Predictive Coding neural network. The transpose of each of the artificial neural network's weight matrices are used for $W^{(i)}$, the Predictive Coding feedback weight matrices. Finally, I simulate the Predictive Coding equations in the discriminative mode of operation with an ambiguous input fixed. I run this setup multiple times starting from many different initial weights.

The results are similar to a Predictive Coding neural network trained without weight

decay. As before, the error nodes fail to converge to zero after ten simulation seconds. The sum of the squares of the error nodes after ten thousand time steps falls between 0.0196 and 0.1375. This is comparable to the 0.0428 found for Predictive Coding weights trained without decay earlier. The maximum eigenvalue found for the Jacobian of its equilibrium was -0.0185, which is also quite high. Now, what happens when we apply weight decay during backpropagation training?

After training with a decay of 0.003, the sum of the squares of the error nodes after ten thousand time steps in discriminative mode becomes a value between 0.0001 and 0.0004, a major improvement over training without decay. With this decay value, the maximum eigenvalue has also reached -0.5. However, using a decay higher than 0.007 makes the network fail to learn the classification task.

We must note that the weight decay used in backpropagation training does not have the same units as the weight decay used in Predictive Coding training, as the former is an update per batch while the latter is an update for each time step. We will generally need to use a lower weight decay value for backpropagation training than the values we use for λ_M and λ_W in Predictive Coding training.

What might be responsible for any differences between backpropagation-trained weights imported into PC networks and Predictive Coding-trained weights?

One simplistic explanation for why differences can emerge is that Predictive Coding training integrates many tiny, potentially opposing values across its state nodes, error nodes, and weights over time. By updating weights in small increments over thousands of time steps, Predictive Coding explores the optimization landscape differently than back-

propagation would. Predictive Coding may cross a different set of pivotal bottlenecks [16] that converge into different valleys. I find experimentally that, starting from the same initial weights, the resulting weights after training for Predictive Coding and backpropagation are indeed different.

Chapter 4

Generating Samples From Classes

The presence of feedback connections in Predictive Coding networks suggests generative capabilities [66]. What I mean by generative capabilities is that the network should be able to generate inputs when provided the class labels of a dataset. This should be possible because the feedback connections in predictive neural networks carry information from the output layer of the network to the input layer. Since I am using a reversed hierarchy, the class labels are provided to layer (0) of the network, the inputs are provided to layer $(n-1)$ of the network, and feedback connections send information about the class from layer (0) up to layer $(n-1)$.

Why does this specific generative capability matter? Generative networks have the potential to be better at discriminative tasks because they can represent the different features of inputs before classification occurs [23]. Currently, autoencoders can generate inputs from latent space [3], but not from a class label alone. The original conception

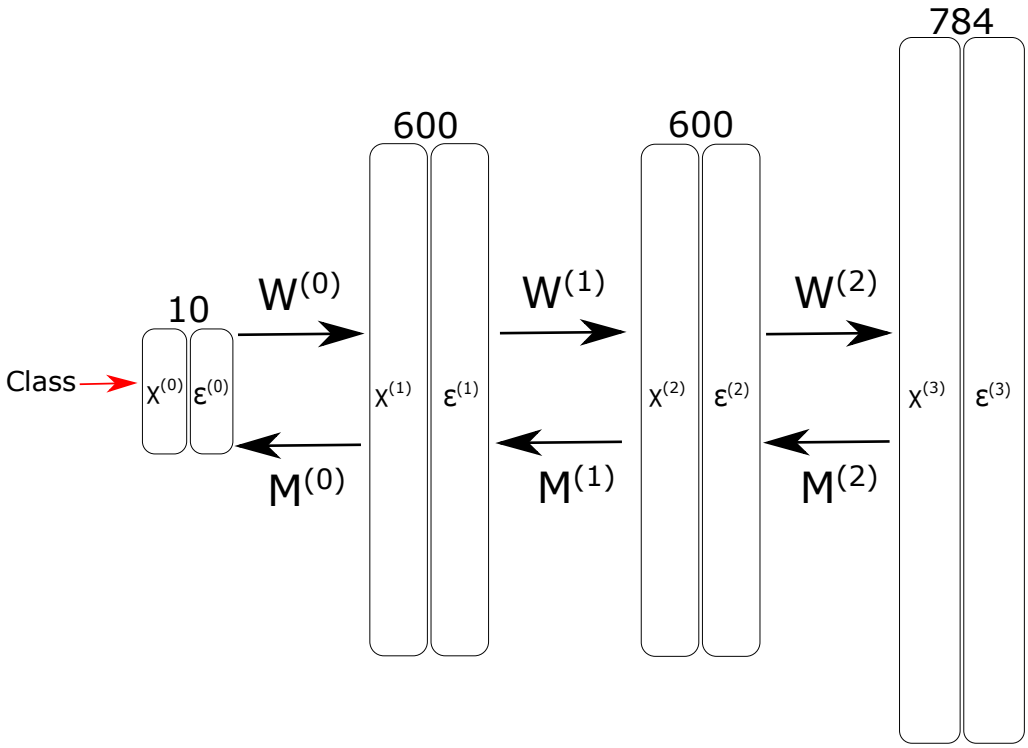


Figure 4.1: An illustration of the **generative** mode of operation in a 10-600-600-784 Predictive Coding network for MNIST. The class vector is fixed to $x^{(0)}$ and the network equations are allowed to run to equilibrium. At the equilibrium, the generated image is contained in the state nodes $x^{(3)}$. The *stim* container is not included.

of Predictive Coding focused on regenerating natural images that the network has been exposed to from a set of latent causes [47], but it also does not generate from a class label. A class label is often smaller and one-hot, containing far less information than a latent space. Priming each layer of the Predictive Coding network with states consistent with the desired input can also help it generate inputs [21], but this still does not address the challenge of generating inputs from a class alone.

To start, I train a 10-600-600-784 PC network with logistic activations on MNIST with Adam [28, 66] to achieve 98% test accuracy on the MNIST dataset. Then, I investigate if the trained network can generate MNIST images from the class labels. To do this, I clamp the MNIST class vectors to the bottom layer of the network, run the network in **generative** mode, and examine the image generated at the state nodes in the top layer. The generative process for this network is shown in Fig. 4.1.

Recall that in the generative mode of operation, $\alpha^{(i)} = 1$ for $1 \leq i \leq n-1$, and $\beta^{(0)} = 0$, $\beta^{(n-1)} = 0$, $\beta^{(j)} = 1$ for all $1 \leq j \leq n-2$. The setting of all α to 1 allows information to flow steadily up the network from the class label. Setting $\beta^{(0)} = 0$ prevents updates to the state nodes $x^{(0)}$ on layer (0) where the class is contained. The input is generated in the state nodes $x^{(n-1)}$, so setting $\beta^{(n-1)} = 0$ prevents $x^{(n-1)}$ from receiving top-down information. Thus, in Fig. 4.1, the error nodes at the top layer, $\varepsilon^{(3)}$, would not send any top-down information to $x^{(3)}$, as $x^{(3)}$ is where the image is being generated.

The results of input generation from MNIST class vectors are shown in Fig. 4.2. It turns out that the trained network generates images that do not look like digits at all. Why is that?

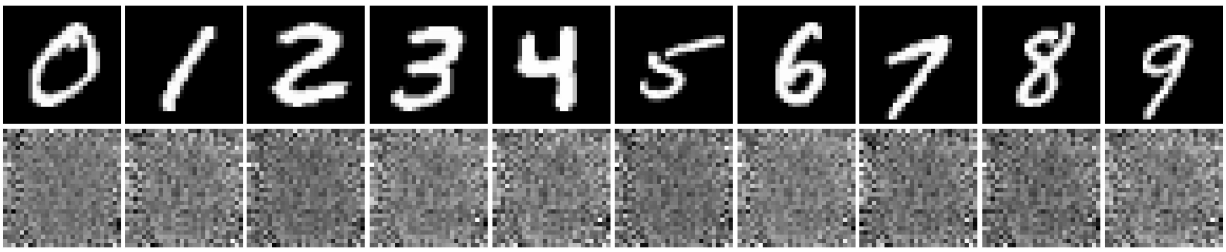


Figure 4.2: Images generated by the network trained on MNIST, taken from [66]. The top row shows a sample of each digit class, while the bottom row shows the corresponding generated image of that class. These generated images do not resemble actual digits.

4.1 The Generative Process Solves An Under-determined System

Consider a Predictive Coding network with two layers in which the input layer has m nodes, the output layer has n nodes, and the dataset has r different classes, with $r \leq n < m$. In other words, there are more input nodes than output nodes, and so more distinct inputs than classes. Thus, the feedforward weight matrix, $M^{(0)}$, has dimensions $n \times m$. Figure 4.3 illustrates an example in which $m = 3$ and $n = 2$.

Next, consider the state of the network after the end of training. The goal of training is to learn an $M^{(0)}$ that maps inputs X to output classes Y. When the network has learned this mapping, then the relation

$$Y = M^{(0)}\sigma(X) \tag{4.1}$$

must be satisfied. It follows that during the discriminative mode of operation, the network nodes will change until

$$x^{(0)} = M^{(0)}\sigma(x^{(1)}). \tag{4.2}$$

Note that (4.2) follows from the equilibrium of (3.4) and (3.1) by setting *stim* to the input X as well as $\varepsilon^{(0)}$ and $\varepsilon^{(1)}$ to 0. Recall that we can set $\varepsilon^{(0)}$ and $\varepsilon^{(1)}$ to 0 since they will equal 0 at the equilibrium in the discriminative mode.

In the generative mode of the network, the class labels are fixed to $x^{(0)}$, so $x^{(0)} = Y$, as

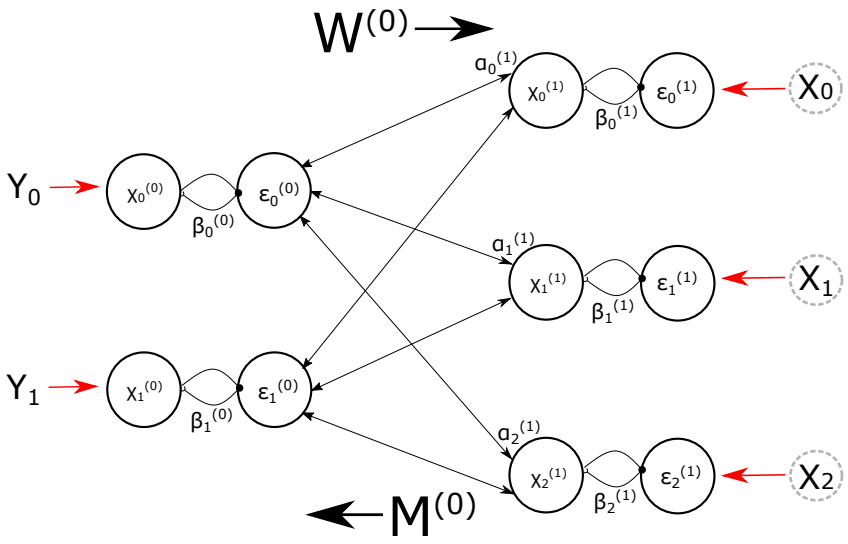


Figure 4.3: Small PC Network with $m = 3$, and $n = 2$.

well as $\alpha^{(1)} = 1$, $\beta^{(1)} = 0$, and $\beta^{(0)} = 0$. The resulting system of differential equations is

$$\tau \frac{dx^{(1)}}{dt} = W^{(0)}\varepsilon^{(0)} \odot \sigma'(x^{(1)}) \tag{4.3}$$

$$\varepsilon^{(1)} = 0 \tag{4.4}$$

$$x^{(0)} = Y \tag{4.5}$$

$$\tau \frac{d\varepsilon^{(0)}}{dt} = x^{(0)} - M^{(0)}\sigma(x^{(1)}) - \nu^{(0)}\varepsilon^{(0)} \tag{4.6}$$

At equilibrium, (4.3) equals zero. As long as $\sigma'(x^{(1)}) \neq 0$, it must be that $W^{(0)}\varepsilon^{(0)} = 0$. Now notice that $W^{(0)}$ has dimensions 3×2, and $\varepsilon^{(0)}$ is a vector of length 2. These dimensions imply that $\varepsilon^{(0)} = 0$ because the matrix system is over-determined. Thus, the first three equations are all solved, and the only remaining unsolved constraint on the equilibrium

comes from (4.6). Setting (4.6) to equilibrium state, where $\varepsilon^{(0)} = 0$, gives

$$M^{(0)}\sigma(x^{(1)}) = Y \tag{4.7}$$

From before, we know that $x^{(1)} = X$ is a solution. However, is this the only solution? Even though the network quickly converges to a state where $x^{(1)}$ satisfies (4.7), and the error nodes reach very small values, I find experimentally that $x^{(1)}$ is typically not close to X [66].

Using the fact that M is a 2×3 matrix and $x^{(1)}$ is a 3-vector, if M is fixed, then we are solving for the 3 variables of $x^{(1)}$ from 2 equations. The process of finding $x^{(1)}$ thus involves solving an *under-determined* system with an infinite number of possible solutions for $x^{(1)}$.

To further understand the problem, we can consider a linear network where all activation functions are the identity function.

4.2 Analysis of Generative Linear Networks

In a linear network, all activation functions $\sigma(x) \equiv x$. Thus, (4.7) becomes

$$M^{(0)}x^{(1)} = Y\,. \tag{4.8}$$

Suppose that $x^{(1)} = \bar{x}$ is a solution to (4.8). Then, for any scalar c, $x^{(1)} = \bar{x} + c\hat{x}$ is also a solution if $\hat{x} \in \text{null}(M^{(0)})$ (i.e. $\hat{x}$ is in the nullspace of $M^{(0)}$). Since (4.8) is under-

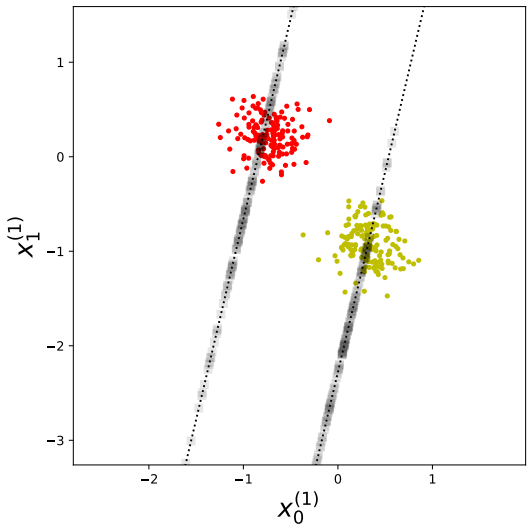


Figure 4.4: A demonstration of generated samples by Dr. Orchard, and the corresponding solution space, taken from [66]. The clusters of red and yellow dots show the training samples for the two classes. The shaded squares show the 300 generated samples, each started from a different initial network state. Note that the figure depicts a 2-D projection of a 3-D space, with the x-axis corresponding to the 0-dimension and the y-axis corresponding to the 1-dimension of the state nodes at the input layer.

determined, we know that a non-trivial $\hat{x}$ exists that solves (4.8). In other words, this linear network has an infinite number of $x^{(1)}$ states that yield zero error nodes at equilibrium. Experimentally, I find that the vast majority of these states correspond to input samples $x^{(1)}$ that do not resemble inputs from the training set.

Dr. Orchard demonstrated the non-uniqueness of solution states $x^{(1)}$ in Fig. 4.4 for the network shown in Fig. 4.3. The network was initialized with random values in the state and error nodes. Then, we set the class vector Y to either $[1, 0]$ or $[0, 1]$ and run the network in generative mode to equilibrium with the appropriate α and β parameters. Each class vector generates a different sample, and the spread of those samples is shown in Fig. 4.4.

The solution spaces are shown as dotted lines, and both pass through the input clusters. The black squares along the dotted lines are the samples generated by the network. Most of them do not fall within the input clusters, even though the network's equilibrium state yields very small values in the error nodes $\varepsilon^{(0)}$ [66].

It is looking bad for the generative mode of operation of Supervised Predictive Coding. Fortunately, hope is not yet lost. The following theorem, proven by Dr. Orchard [43, 66], shows that linear networks can still be generative.

Theorem 1. *Given a matrix of r linearly-independent m-vectors,*

$$X = [X_1|\cdots|X_r] \in \mathbb{R}^{m\times r}$$

and a corresponding matrix of n-vectors,

$$Y = [Y_1|\cdots|Y_r] \in \mathbb{R}^{n\times r}$$

with $r \leq n < m$, there is an $n \times m$ matrix,

$$A = \left[\begin{array}{c} A_1 \\ \hline \vdots \\ \hline A_n \end{array}\right]$$

such that the minimum 2-norm solution x^* to $Ax = Y_i$ is $x^* = X_i$. Moreover, the jth row of A is the minimum 2-norm solution of $aX = Y$ for $a \in \mathbb{R}^{1\times m}$.

The proof of this theorem is written in Appendix A.

Using the theorem, it follows that applying a minimum 2-norm constraint to (4.8) collapses the solution spaces for M and x to unique solutions, and the unique solution for x will be close to the training input. To apply the theorem to our neural networks, we do two things. First, during training, we solve for the rows of $M^{(0)}$ by finding the minimum 2-norm solution of

$$M^{(0)}X = Y$$

Once $M^{(0)}$ is found, we can generate an input sample corresponding to the output class vector Y_i by finding the minimum 2-norm solution $x^{(1)}$ of

$$M^{(0)}x^{(1)} = Y_i\,.$$

4.2.1 Weight and Activity Decay

Dr. Orchard's proof of Theorem 1 shows that the minimum 2-norm solution can be attained using singular value decomposition. Another way to approximate the minimum 2-norm solution is to solve the system defined by the Predictive Coding equations iteratively while including a term in the objective function that penalizes for the 2-norm of the solution.

Suppose the linear system $Ax = y$ is under-determined, so A has more columns than rows. We can solve that linear system by minimizing the 2-norm error [66],

$$\min_x \|Ax - y\|_2^2\,.$$

As long as A is full-rank, that is its rows are linearly independent, there is a non-zero x that yields an error of zero. Adding the penalty term for the 2-norm of x gives

$$\min_x \left[\|Ax - y\|_2^2 \; + \; \lambda \|x\|_2^2 \right],$$

where $\lambda > 0$ is a regularization constant that sets the weight of the penalty term. Solving this optimization problem by gradient descent yields the updates,

$$\frac{dx}{dt} \propto -A^T (Ax - y) - \lambda x \tag{4.9}$$

In the context of our neural network, the penalty term can be applied to both $x^{(1)}$ and $M^{(0)}$ simultaneously [66]. Importantly, notice that this update is very similar to the Predictive Coding equations with weight decay applied, (3.10), (3.12), and (3.13), that had been derived for stability analysis. The equations are shown again below fitted to the two-layer network used for our analysis.

$$\tau \frac{dx^{(1)}}{dt} = \alpha^{(1)} W^{(0)} \varepsilon^{(0)} \odot \sigma'(x^{(1)}) - \beta^{(1)} \varepsilon^{(1)} - \boxed{\lambda_x x^{(1)}} \tag{4.10}$$

$$\gamma \frac{dM^{(0)}}{dt} = \varepsilon^{(0)} \otimes \sigma(x^{(1)}) - \boxed{\lambda_M M^{(0)}} \tag{4.11}$$

$$\gamma \frac{dW^{(0)}}{dt} = \sigma(x^{(1)}) \otimes \varepsilon^{(0)} - \boxed{\lambda_W W^{(0)}} \tag{4.12}$$

Compare (4.9) with (4.10). A and A^T are replaced by M and W, $\varepsilon^{(0)}$ is exactly $(Ax - y)$, $\beta^{(1)} = 0$, and the similarity between the decay terms is obvious.

Dr. Orchard demonstrates in Fig. 4.5 the same experiment whose results were displayed

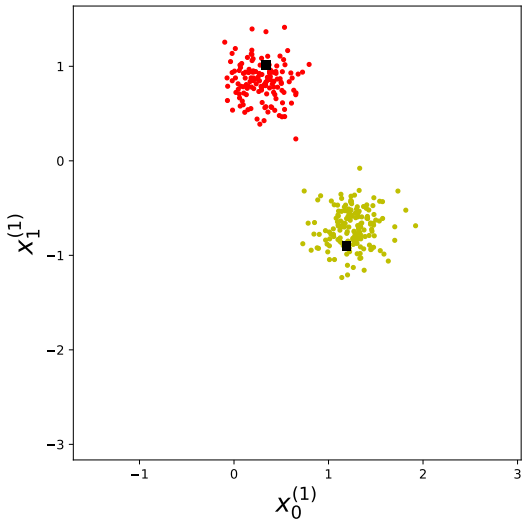


Figure 4.5: Generated samples using decay by Dr. Orchard, taken from [66]. Compare this figure to Fig. 4.4. The black squares are actually many shaded squares superimposed on top of each other. Note that the plot depicts a 2-D projection of a 3-D space, with the x-axis corresponding to the 0-dimension and the y-axis corresponding to the 1-dimension of the state nodes at the input layer.

in Fig. 4.4, but using a weight decay of 0.05 for $M^{(0)}$ and $W^{(0)}$ during training and an activity decay of 0.05 for the state nodes $x^{(1)}$ during input generation. Note that $W^{(0)}$, the feedback weight matrix, is very important to the generative mode. It uses the information in the error nodes $\varepsilon^{(0)}$ to update $x^{(1)}$, pushing those state nodes towards the appropriate set of values that would allow $M^{(0)}$ to map them down to the fixed class vector at $x^{(0)}$. Since $\varepsilon^{(0)}$ eventually goes to 0, this update gradually gets smaller and smaller until $x^{(1)}$ stabilizes to the values of the generated samples.

Notice that the generated samples (the black squares) in Fig. 4.5 are much closer to the cluster centroids than the samples generated by the PC-non-decay network, depicted in Fig. 4.4.

4.2.2 Deep Linear and Tanh Networks

Can adding decay to the Predictive Coding equations improve the generative capabilities of deep Predictive Coding networks?

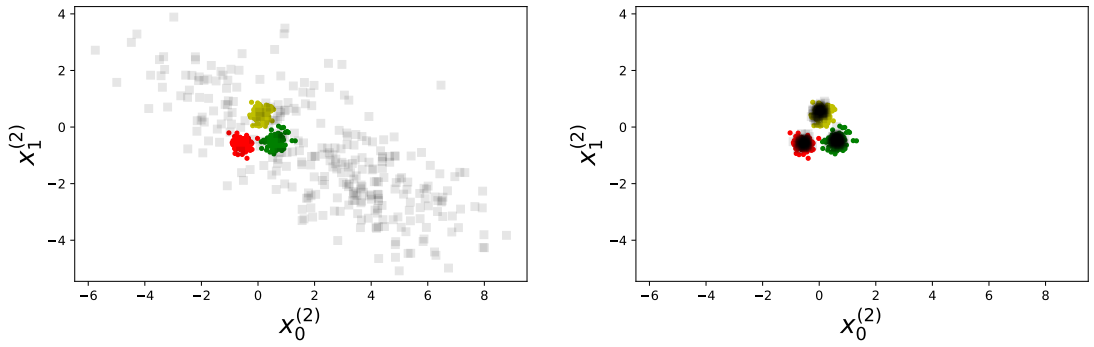
The answer is yes [66]. In the next couple of sections, I will show on a dataset that decay benefits both the training and input generation modes of Predictive Coding networks.

To do this, Dr. Orchard and I created a small dataset consisting of three 10-D vectors, each created by drawing 10 uniformly-distributed random numbers from the range $[-1, 1]$. These three vectors acted as the exemplars v for each of three classes. The classes were one-hot vectors in 3-D. A dataset of 300 training samples was created by adding Gaussian noise (mean of 0, standard deviation of 0.1) to the class exemplars.

Experimental Setup: Dr. Orchard and I used the dataset to train a network with 3 output nodes, 5 hidden nodes, and 10 input nodes (3-5-10). For each trial, we trained our network for five epochs, fixing each input/target pair for 4 seconds to allow the network to hopefully reach equilibrium. We set τ to 0.05 seconds, and γ to 0.1 seconds.

After training, we ran the network in generative mode for 10 simulation seconds on all 300 of the one-hot class vectors in the dataset, starting from a random initial network state each time; the initial x and ε values were drawn from a Gaussian distribution with mean 0 and a standard deviation of 1. We performed this training-testing procedure on two networks: one PC-non-decay network ($\lambda_M = \lambda_W = \lambda_x = 0$), and one PC-decay network ($\lambda_M = \lambda_W = 0.05, \lambda_x = 0.01$).

Results: Figure 4.6 shows the generated samples for the two different networks. The samples generated by the PC-non-decay network shown in (a) exhibit a wide dispersion,



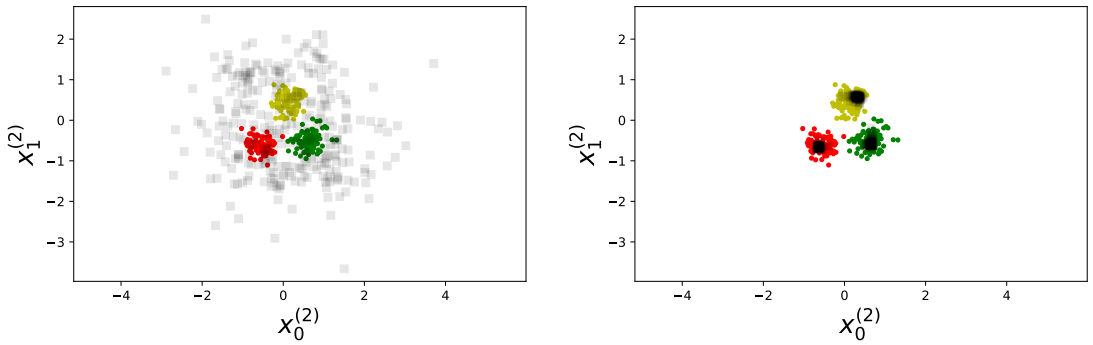
(a) PC-non-decay Linear Network (b) PC-decay Linear Network

Figure 4.6: Dr. Orchard's generated samples, verified by myself. For (a), the PC-non-decay network was trained without weight decay, and the samples were generated from the one-hot class vectors without any activity decay. For (b), the PC-decay network was trained using weight decay of 0.05, and the samples were generated from the one-hot class vectors using an activity decay of 0.01. Note that the plot depicts a 2-D projection of a 10-D space, with the x-axis corresponding to the 0-dimension and the y-axis corresponding to the 1-dimension of the state nodes at the input layer.

while those generated by the PC-decay network are tightly packed close to the centre of the corresponding cluster.

Even though the theorem is technically only valid for linear networks, we were interested to see if the decay also helped nonlinear networks generate samples that were similar to the training inputs. We re-ran the above experiment (with the 3-5-10 architecture), but using tanh activation functions. The results are shown in Fig. 4.7. Again, the PC-decay network generated input samples that were much more similar to the training inputs than the PC-non-decay network.

The next two experiments were designed to isolate the effect of each type of decay on generative PC networks. To start, we will show that activity decay significantly reduces



(a) PC-non-decay Tanh Network (b) PC-decay Tanh Network

Figure 4.7: Generated samples by tanh 3-5-10 networks on the same dataset as that used in Fig. 4.6. For (a), the PC-non-decay network was trained without weight decay, and the samples were generated from the one-hot class vectors without any activity decay. For (b), the PC-decay network was trained using weight decay of 0.05, and the samples were generated from the one-hot class vectors using an activity decay of 0.01. Note that the plot depicts a 2-D projection of a 10-D space, with the x-axis corresponding to the 0-dimension and the y-axis corresponding to the 1-dimension of the state nodes at the input layer.

the variance of generated samples.

4.2.3 Activity Decay Generates Unique Samples

For this experiment, Dr. Orchard and I consider three different linear networks: an untrained network, a network trained without weight decay, and a network trained with weight decay.

Experimental Setup: For the untrained network, we randomly assigned weights in $M^{(i)}$ using a Gaussian distribution with 0 mean and standard deviation of $1/(2\sqrt{N})$ where N is the number of nodes in layer i. We then set $W^{(i)}$ to be the transpose of $M^{(i)}$ to ensure that the products $M^{(i)}W^{(i)}$ were symmetric positive semi-definite. For the decay trained

Table 4.1: Average standard deviation of generated samples

Network	No Activity Decay	Activity Decay
Untrained	0.796	0.0048
Trained without Weight Decay	0.817	0.0048
Trained with Weight Decay	0.823	0.0055

network, we used a decay of $\lambda_M = \lambda_W = 0.05$. Both trained networks classified the dataset with 100% accuracy.

Each of those three types of networks was tested in generative mode, starting with a random state in which the initial x and ε values were drawn from a Gaussian distribution with mean 0 and standard deviation of 1. We clamped the bottom layer, $x^{(0)}$, to a chosen class vector and ran the network for 10 seconds with a time step of 0.001 seconds. After generating 100 samples for each of the 3 classes, we recorded the standard deviation of each of the 10 input nodes, $x^{(2)}$. Since there are 3 classes and 10 input nodes, we tabulated 30 standard deviations for each network. We then computed the mean of the 30 standard deviations and used that as a measure of variation in the network's generated samples.

The generative part of this experiment was performed under two different conditions. In the first condition, the networks were run in generative mode without activity decay ($\lambda_x = 0$). In the second condition, the networks were run with an activity decay of $\lambda_x = 0.05$.

Results: The results are shown in Table 4.1. For all three networks, applying activity decay reduces the standard deviation of the generated samples by orders of magnitude. Moreover, when using activity decay, the standard deviation of the generated samples

asymptotically approaches zero. That is, the generated samples converge on unique solutions when using activity decay.

But are the generated samples similar to the training inputs? That is the topic of the next experiment.

4.2.4 Weight Decay Benefits Generative Quality

While activity decay helps the generative phase converge to a unique solution, that solution may not resemble the inputs from the training dataset. In this section, Dr. Orchard and I show that weight decay during training is a key factor that enables PC networks to generate samples close to the training inputs.

Experimental Setup: I trained a linear PC-decay network ($\lambda_M = \lambda_W = 0.05$) and a linear PC-non-decay network ($\lambda_M = \lambda_W = 0$) on the dataset from the previous section. Both types of networks also used the 3-5-10 architecture described in the previous section. During training, I used $\tau = 0.02$ and $\gamma = 0.1$, I clamped each input/target pair for 5 seconds, used a time step of 0.001 seconds, and trained for 10 epochs with a batch size of 20. I repeated the training starting from 10 different initial sets of weights. The weights for $M^{(i)}$ and $W^{(i)}$ were each sampled from a Gaussian distribution with 0 mean and standard deviation of $1/(2\sqrt{N})$, where N is the number of nodes in layer i. This process gave us 20 networks in total, 10 trained with weight decay, and 10 trained without weight decay.

After training, the class vectors were fixed to each network while they were run in generative mode with a time step of 0.001 seconds and an activity decay of $\lambda_x = 0.01$. To improve stability in the non-decay-trained networks, τ was set to 0.2 since these networks

Table 4.2: Euclidean distances of generated samples

Network	Minimum	Average	Maximum
Weight Decay Training	0.2216	0.4076	0.6333
No Weight Decay Training	5.032	100.9	691.8

had large weight matrix norms. The decay-trained networks did not require increasing τ in order for them to be stable. To verify equilibrium, I checked that $\frac{dx^{(i)}}{dt} < 0.001$ and $\frac{d\varepsilon^{(i)}}{dt} < 0.001$ for all layers i in the decay networks. The non-decay networks were sometimes unstable even with the increased τ. For these unstable networks, I stopped running them after 10.0 seconds.

I computed the Euclidean distance of each generated sample to its corresponding prototype, the centroid of each class. For each network type (i.e. trained with weight decay or trained without weight decay), I generated 1 sample for each of the 3 classes, yielding a total of 30 generated samples per network type. I recorded the average, minimum, and maximum distances across the 30 generated samples.

Results: The results are shown in Table 4.2. The networks trained with weight decay generated samples that were much closer to the class prototypes than the non-decay networks. Their generated inputs yielded distances less than 1.0, placing the generated samples well within the distribution of training inputs, like that shown in Fig. 4.6(b). The networks trained without weight decay generated samples that were unique (thanks to activity decay), but much further from the class prototypes.

4.2.5 Are Backpropagation-trained Networks Generative?

Are artificial neural networks trained by backpropagation with weight decay also generative?

Experimental Setup: I trained a 3-2 neural network with identity activations using a learning rate of 0.01, weight decay of 0.01, and no bias for 500 epochs with a batch size of 30 on a similar dataset as that shown in Fig. 4.4 and Fig. 4.5. It achieved 100% classification accuracy.

I imported the learned weights into a 2-3 Predictive Coding network and ran the network in generative mode with an activity decay of $\lambda_x = 0.01$ for 10 simulation seconds with $\tau = 1.0$ and a time step of 0.001.

Results: The results are shown in Fig. 4.8. The average Euclidean distance between the generated samples and the exemplar vectors is 0.7976, which is nearly double that of the average from using Predictive Coding with weight decay in Table 4.2. Visually, we see that the generated inputs are slightly distant from the cluster centroids. This result is fairly consistent across 10 different datasets and across a range of sensible weight and bias decay values of between 0 and 0.05.

As discussed earlier, Predictive Coding training does many weight updates across thousands of time steps. This training strategy means that PC nets may explore the optimization landscape differently than backpropagation would. In this experiment, it would seem that Predictive Coding training with weight decay outperforms backpropagation training with weight decay at attaining weights that can generate an accurate sample from the class vectors.

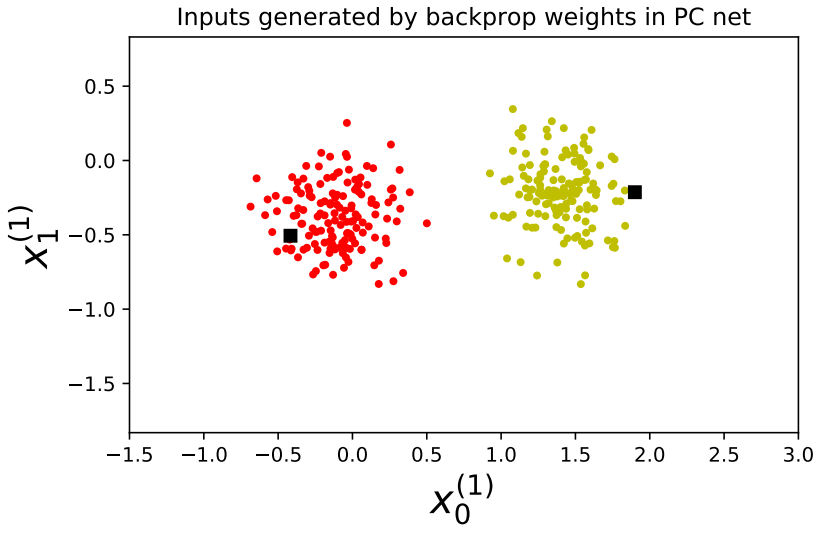


Figure 4.8: Inputs generated from classes by a PC network with weights imported from a backpropagation-trained neural network with decay. Note that this is a 2D projection of a 3D space, with the x-axis corresponding to the 0-dimension and the y-axis corresponding to the 1-dimension of the state nodes at the input layer.

4.2.6 Generating on Multimodal Datasets

So far, the experiments show that PC networks can generate good quality inputs on datasets where each class corresponds to one Gaussian cluster. On datasets where classes are multimodal, can PC networks generate good inputs?

Experimental Setup: To find out, I designed a multimodal Gaussian dataset containing two classes, shown in Fig. 4.9. One class corresponds to the red dots and one class corresponds to the yellow dots. The red class is associated with two Gaussian clusters, while the yellow class is associated with one Gaussian cluster. Using this dataset, I trained two networks with 10 input nodes, 5 hidden nodes, and 2 output nodes using backpropagation. One network was trained without decay, and another network was trained

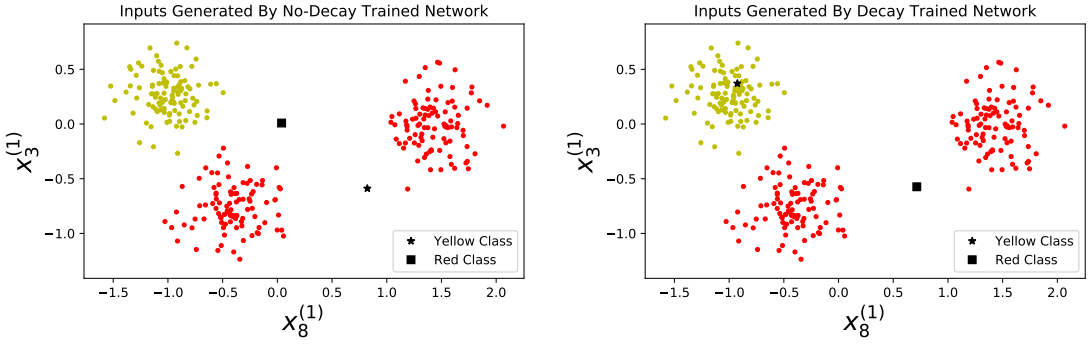


Figure 4.9: Inputs generated by networks trained on a multimodal dataset where one of the classes corresponds to two clusters. The square is an input generated from the red class vector, while the star is an input generated from the yellow class vector. The x-axis is the value of the state node $x_8^{(1)}$, while the y-axis is the value of the state node $x_3^{(1)}$.

with a backpropagation weight decay of 0.004. Both networks used linear activations and started from the same initial weights. I used backpropagation instead of Predictive Coding training because the goal of this experiment is to show the effect of decay training on a multimodal dataset, so the type of learning algorithm used does not matter much as long as the algorithm facilitates the use of decay. Both networks were trained for 20 epochs with a learning rate of 0.01 and a batch size of 4, attaining 100% classification accuracy.

After backpropagation training, the weights were imported into a 2-5-10 Predictive Coding network with $\tau = 0.2$. The networks were simulated for 20 seconds using a time step of 0.001 and an activity decay of $\lambda_x = 0.01$ in generative mode with the red and yellow class vectors fixed to the networks.

Results: The generated samples are shown in Fig. 4.9. The inputs generated by the network trained without decay are not close to the true clusters. For the decay-trained network, notice that it generates a good input sample for the yellow class, but chooses a

mean point between the two clusters for the red class. This result is consistent across ten different trials, and it also occurs with Predictive Coding training.

When a class is associated with more than one cluster, Predictive Coding generates samples between those clusters rather than samples that land in one of the clusters. This property is a result of the fact that Predictive Coding assumes that inputs of each class come from a unimodal Gaussian distribution. Therefore, Predictive Coding networks will associate each class with a unimodal Gaussian even if that class is multimodal. The mean of the unimodal Gaussian will be a point between all the Gaussian clusters of that class.

I have observed that in datasets with multimodal classes *and* input clusters that are not all linearly separable, Predictive Coding networks struggle to succeed at both generating good quality inputs and classifying perfectly. In order to get the network to generate good samples from the classes on these datasets, we are sometimes forced to use a weight decay value that prevents the network from attaining 100% classification accuracy.

Our next set of experiments focus on MNIST. Like in the multimodal dataset, the MNIST digits do not seem to be from unimodal Gaussian distributions. I expect that, for each MNIST class, the network will generate an intermediate image, reflecting a blend of various forms present in the digit class. MNIST is also not completely linearly separable. Thus, I expect that training with a weight decay that allows our network to generate good images may result in a trade-off in classification accuracy.

4.3 Generating MNIST Digits

Now, we can revisit the MNIST dataset to see if adding decay allows the network to generate digit-like samples.

Note that the major issue with MNIST is that it is not an invertible Gaussian dataset since each digit cannot be delineated into separable Gaussian clusters. Additionally, by using the non-linear tanh function in my neural networks during training, Theorem 1's preconditions are violated and so its results may not apply. To recap, Theorem 1 states that it is possible to obtain the unique, minimum norm weights that map X to Y on linear, single-layer networks. Nevertheless, I want to see if decay makes Predictive Coding networks attain good generative capabilities on MNIST.

4.3.1 Normalized Correlations Using Tanh

Experimental Setup: I trained two types of 10-600-784 Predictive Coding networks with tanh activations on 1000 digits of MNIST. One network type was trained using weight decay ($\lambda_M = \lambda_W = 0.05$), and the other network type did not use any weight decay ($\lambda_M = \lambda_W = 0$). I used time constants of $\tau = 0.2$ and $\gamma = 0.8$ for both types of networks. This was repeated over 20 trials. Each pair of decay/non-decay networks was initialized with the same weights, so 20 sets of initial weights were used in total.

After training, each network generated a sample for each of the 10 digit classes, starting from a network state in which all activities were set to zero. The samples were generated by running the Predictive Coding equations for each network for 60 simulation seconds

using a time step of 0.002 seconds with an activity decay of 0.05.

To quantify the quality of the generated samples, I used the normalized correlation between the generated sample x and an exemplar vector v, also known as Cosine Similarity,

$$\text{Corr}(x, v) = \frac{x \cdot v}{\|x\|\|v\|} .$$

The exemplar vector for this experiment is the mean MNIST digit of the corresponding class. Since there are 10 different digits, 2 types of networks, and 20 trials, this yields 400 cosine similarity values, 200 for each network type.

Results: Fig. 4.10 shows a histogram of the cosine similarities. The networks trained without decay generate images that display much lower cosine similarities, in the range of $[-0.1, 0.2)$, with the majority of them being around 0.0, indicating no correlation with the mean digit of each class. The networks trained with decay generate images with cosine similarities in the range of $[0.2, 0.7]$. This suggests they are strictly better at image generation from the class vector than the non-decay networks.

Table 4.3 displays the average cosine similarity and maximum cosine similarity for the decay-trained and non-decay-trained networks. The average similarity of 0.0098 for the images generated by the non-decay networks is consistent with the noisy images we observed in Fig. 4.2. Those images exhibit virtually no identifiable relation to digits.

The images generated by the decay-trained networks exhibit some correlation with the mean MNIST digit with an average of 0.4274, though this is not very close to 1.0. One reason for this could be the high dimensionality of MNIST, at 784 dimensions, compared to just the 10 dimensions I had used for the linear 3-5-10 network experiments. It is much

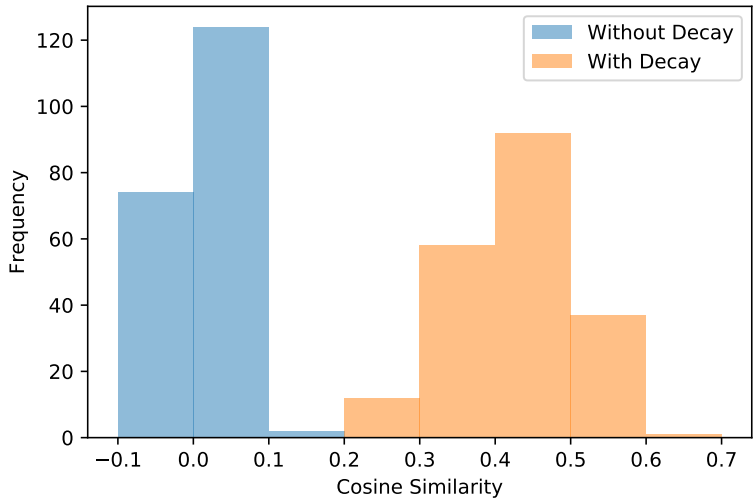


Figure 4.10: Histogram of cosine similarities between the PC-net generated images and the mean MNIST digit of each class

more difficult to arrive at the precise solution when solving an under-determined system with hundreds of free variables. Another reason could be that I only trained on 1000 randomly chosen digits of MNIST rather than the full training dataset. Since the networks were not exposed to the full dataset, they could not really learn the weights that would map to the states close to the mean of each MNIST digit. A third reason could be the use of tanh activation functions in my network, which is nonlinear.

Table 4.3: Average and maximum cosine similarities of PC net-generated MNIST digits

Network	Average Cosine Similarity	Max Cosine Similarity
Non-Decay Net	0.0098	0.1170
Decay Net	0.4274	0.6081

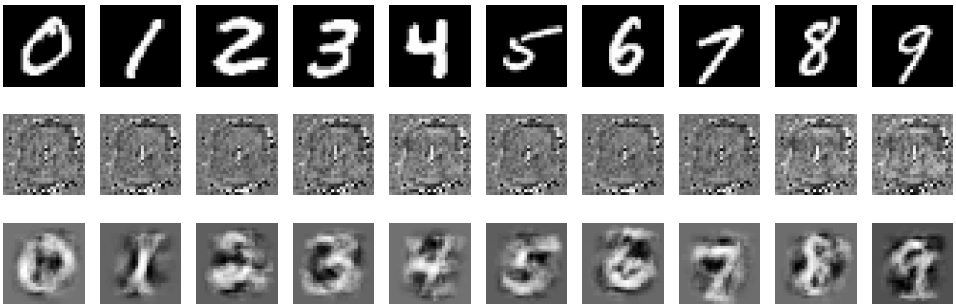


Figure 4.11: Generated samples, taken from [66]. (top row) Sample digits from the training set. (middle row) Samples generated without decay. (bottom row) Samples generated with weight and activity decay.

4.3.2 Generating The Best Looking Digits

The previous experiment only trained on 1000 MNIST digits. I am interested in examining how the networks would fare at image generation if trained on the entire MNIST training set of 50,000 digits.

Experimental Setup: Dr. Orchard and I trained two 10-600-600-784 Predictive Coding networks for 10 epochs on the full training set of 50,000 MNIST samples. One was trained with decay, and the other was trained without decay. To be exact, the network used tanh activation functions for each layer, had 784 input nodes, two hidden layers with 600 nodes each, and an output layer with 10 nodes. After training, we fixed the 10 class vectors to each network and ran them in generative mode. We used the parameters $\tau = 0.08$, $\gamma = 0.8$, $\lambda_M = \lambda_W = 0.05$, and ran them in generative mode with $\lambda_x = 0.001$ for 60 seconds at a time step of 0.002.

Results: Fig. 4.11 shows samples generated by a PC network trained without decay,

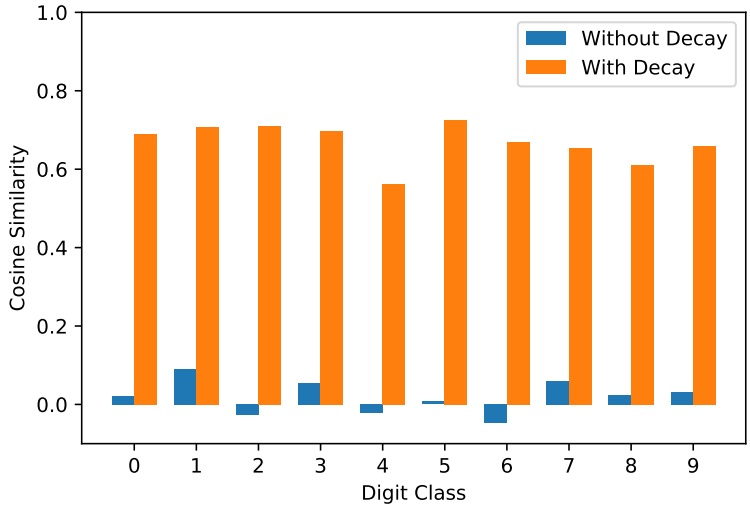


Figure 4.12: Bar graph of cosine similarities between the PC-net generated images and the mean MNIST digit for each digit class

along with samples generated by a decay-trained network. Even though the networks used a nonlinear activation function, the network with decay generates samples that resemble digits. In addition, the decay network achieved 85% accuracy within the top two classes.

Fig. 4.12 plots the cosine similarity individually for each of the ten digit classes. As before, the similarity is measured relative to the average MNIST digit for each class. The average cosine similarity across the ten classes for the no-decay net is 0.019 and 0.6682 for the decay net.

The decay net's images have much higher cosine similarities for each class than the non-decay net's. This result is consistent with the visual observation that the decay net's generated digits shown in Fig. 4.11 are much closer to what each class's digit looks like than the non-decay net's. The decay net's digits are recognizable, but the non-decay net's

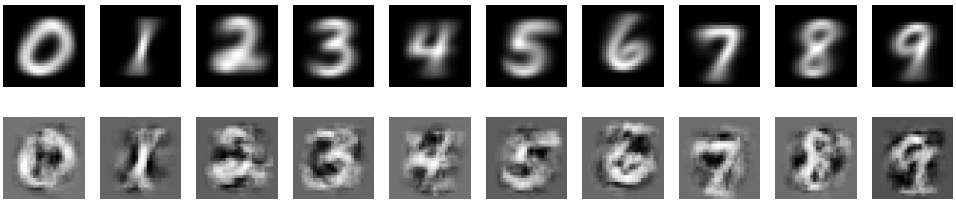


Figure 4.13: Mean MNIST digits (top row). Samples generated by the decay-trained PC network (bottom row).

digits look like noise. The noisy images have almost no observable correlation with the mean digits of each class.

As an interesting observation, the top row in Fig. 4.13 shows what the mean MNIST digits actually look like. They are somewhat wispy and look surprisingly similar to the images generated by the decay net, only with starker contrasts. This visual similarity may arise from the fact that a PC network implicitly associates each class to a Gaussian distribution. The mean of a class's distribution is close to the average across all the inputs for that class. In other words, the PC-decay network constructs a Gaussian for each class, with a mean close to the average digit for that class.

The Predictive Coding network thus tends to generate the mean of each digit from the class vector rather than a particular digit.

Chapter 5

Discussion

In this thesis, I have presented Predictive Coding networks as an alternative to back-propagation that achieves similar statistical learning results. Dr. Orchard and I showed that Supervised Predictive Coding is not generative by default, and our analysis pinpoints the reason behind this observation: the generative problem is under-determined, so there are many states consistent with the network constraints that are not necessarily close to the input samples we wish to generate.

Dr. Orchard proved a theorem for linear networks that tells us we can generate samples resembling the training inputs by imposing a minimum 2-norm constraint on the network's weight matrices and on the network's state nodes. This constraint is implemented by adding weight decay to the network's update equations for its forward weights M and feedback weights W during training, and by adding activity decay to the network's update equations for its state nodes x during input generation.

The theorem can be recursively applied for deep linear networks by treating every pair of adjacent layers $x^{(i+1)}$ and $x^{(i)}$ with connection weights A as a linear system $Ax^{(i+1)} = x^{(i)}$. We showed experimentally that adding decay allows deep networks as well as networks with nonlinear tanh activations to become generative. Additionally, training with weight decay pushes all the eigenvalues of the Jacobian of the equilibrium of the network equations in discriminative mode towards -0.5. This has the effect of making the network converge to equilibrium in less time steps when the network classifies an input.

One interesting quality I had mentioned observing about the generated digits is that they do not resemble any particular digit from the training images, but look more like the mean of each digit. This is because, during training, the network learns to associate multiple samples from each class to a one-hot class vector. In order to accomplish this, the network adjusts its synaptic weights such that its weight matrices map multiple different points from the 784-dimensional digit space to the same class. Those learned weights represent a compromise between all the digits it has seen since it needs to associate every one of them with the correct class.

During generation, the class is fixed while the same weights are used to build up images at the input layer, so the generated digits will reflect the compromise that the weights have learned. Without priors on the network's state nodes, the network generates a compromised, average-looking digit rather than a particular digit from the training sample. Perhaps this is similar to the way human brains conceptualize an essential shape for each digit they have learned. Under average conditions, this shape does not change; we prefer to draw a specific 6 rather than a different 6 every time we write down that number. Maybe we tend to visualize the same 6 as well. However, this is philosophical conjecture, and

more data from neuroscience would be necessary to back up this conjecture.

There are a couple more questions worth asking: what are the limitations of these discoveries, and are Predictive Coding networks actually biologically plausible?

5.1 Limitations of Decay

One limitation is that adding a high decay negatively impacts classification accuracy on certain datasets [66]. As discussed earlier, MNIST digits are multimodal and non-linearly separable.

On our constructed, invertible datasets, like the ones shown in Fig. 4.5 and Fig. 4.6, PC-decay networks were able to both generate good quality inputs and classify every data point correctly. Those datasets were made by adding Gaussian noise to a number of class prototypes. But on MNIST, our chosen decay of $\lambda_M = \lambda_W = 0.05$ that allowed the network to generate good digits seemed to exhibit a reduction in accuracy of about 5% to 10%, compared to the same network trained without decay. As we decrease the decay rate, the network climbs in classification accuracy, but generates images that look less like the training images.

The drop in accuracy could also be due to the extra penalties in the objective function,

$$F = -\sum_{i=0}^{n-1} \frac{1}{2}\|\varepsilon^{(i)}\|^2 - \sum_{i=0}^{n-1} \lambda_x \|x^{(i)}\|^2 - \sum_{i=0}^{n-2} \lambda_M \|M^{(i)}\|^2 - \sum_{i=0}^{n-2} \lambda_W \|W^{(i)}\|^2 \tag{5.1}$$

Here, (5.1) is a repeat of (3.9). By including norms of the state nodes and weight matrices in the objective, the network may be sacrificing accuracy for smaller state nodes and lower

weight matrices to achieve a compromise between all the terms in (5.1). One way to fix this could be to lower λ over time. Another way to fix this is to reformulate the objective as a constrained optimization problem in which we minimize $\|x\|^2$, $\|M\|^2$, and $\|W\|^2$ subject to $\|\varepsilon\| = 0$, so that it is a priority that all the error nodes always reach 0 [66].

Whittington and Bogacz used a slightly different algorithm than ours in their code [68] during learning. They alternately update all the state nodes (x) and all the error nodes (ε). This strategy takes advantage of the bipartite graph structure between state and error nodes in Predictive Coding networks, with observable improvements in the speed of the network's convergence to equilibrium [66]. I have not yet investigated how decay would perform if used in this accelerated convergence strategy, but I expect that decay would yield similar results as it did in our continuous-time simulations of the Predictive Coding differential equations. Whittington and Bogacz also implemented Adam [28] in their learning algorithm [68], and work needs to be done to determine how decay would work when using Adam with the predicting coding equations.

Note that Theorem 1 does not mention the feedback weights W. Experimentally, Dr. Orchard and I find that excluding the decay term for W often causes the learning to become unstable, resulting in weights that soar to infinity or infinitesimally small values [66]. Since it is not clear why this happens, it is worth investigating in future work.

For future work, Dr. Orchard and I could investigate the degree to which generative Predictive Coding networks are susceptible to adversarial attacks. Can the internal errors generated by a mismatch between input and classification protect the network from catastrophic misclassification of adversarially-perturbed inputs?

5.2 Biological Plausibility of Predictive Coding

We revisit the four biological plausibility criteria created by Whittington and Bogacz [68] I had used to judge backpropagation.

Local Computation: State and error nodes in Predictive Coding networks perform all their computations, as shown in their update equations (3.1), (3.2), (3.3), and (3.4), using values from the nodes they are directly connected to. This criteria is satisfied even with activity decay, since decay only subtracts the value of the state node itself.

Local Synaptic Plasticity: Synaptic weights in Predictive Coding networks are changed using only values from the state and error nodes the synapse connects, as shown in the weight update equations (3.5) and (3.6). This criteria is satisfied even with weight decay, since decay only subtracts the value of the synaptic weight itself.

Minimal External Computation: The Predictive Coding network can autonomously switch between discriminative, generative, and clamped modes of operation based on whether or not an input is held to the input layer, a class is held to the output layer, or both.

Plausible Architecture: Whittington and Bogacz note that Predictive Coding does not yet fulfill this criteria [68].

First, the one-to-one connections between state and error nodes at each layer have not been observed in the brain [68]. There is, however, evidence that neurons in the visual cortex can encode an error signal [11]. It may be possible that states and errors exist simultaneously in a neuron [20], and the mechanism behind whether a neuron currently

encodes state or error is still unknown.

One interesting direction to continue from here is to remove the one-to-one pairings of state and error nodes in general. Any state node can be connected to any other state or error node through synaptic weights not necessarily equal to 1, and similarly for the error nodes. This direction involves heavy investigation into how the Predictive Coding equations would have to be reformulated to accommodate the removal of one-to-one pairings. The new equations will implicate extensive theoretical and experimental analysis of whether such a network would still be able to achieve statistical learning.

While Whittington and Bogacz's model uses symmetric weights [68], our model uses asymmetric weights, avoiding the weight transport problem [37]. That is a plus for biological plausibility for us!

Another problem with the biological plausibility of Predictive Coding is more subtle and relates to the values encoded within error nodes. Whittington and Bogacz note that error nodes can be either positive or negative, but biological neurons cannot have negative activity [68]. They propose that error nodes are biological rectified linear neurons [6], and that one can associate zero activity in error nodes with the baseline firing rate of those neurons [68].

However, in their words, *such an approximation would require the neurons to have a high average firing rate, so that they rarely produce a firing rate close to 0, and thus rarely become nonlinear* [68]. *Although the interneurons in the cortex often have higher average firing rates, the pyramidal neurons typically do not* [40, 68]. Since much of the cortex consists of pyramidal neurons [27], Predictive Coding may not yet be the right model of

the cerebral cortex.

Another way to fix the problem of negative activity is to assume that error nodes are not singular neurons. Both state and error nodes may be populations of neurons, a subset of which fires inhibitory activities representing negative values, and another subset fires excitatory activities representing positive values. The sum of inhibitory and excitatory activity in the population represents the positive or negative value encoded in an error node.

5.3 Conclusion

Predictive Coding is an insightful, biologically inspired framework of neural computation that can attain backpropagation learning. I have added two forms of decay, weight decay and activity decay, to the supervised learning formulation of Predictive Coding. Weight decay helps to regularize the weight matrices of Predictive Coding networks, allowing the network to converge to equilibrium more quickly during input classification. Combining weight and activity decay allows Predictive Coding networks to simultaneously classify inputs and generate inputs from classes, an achievement that paves the way for future work on bidirectional learning of associations.

Our next step may involve developing new, biologically plausible neural learning algorithms and gathering data in neuroscience to verify that the outputs of that algorithm are consistent with the computations performed in the brain. The destination to artificial general intelligence will also require advances in neuromorphic computing. Currently, my

Predictive Coding model learns much more slowly compared to ANNs trained by backpropagation. Neuromorphic computing leverages the local computation aspects of Predictive Coding networks with hardware modelled after biological neural networks, allowing Predictive Coding to be simulated much more efficiently. Other biologically plausible models of neural computation will also be able to enjoy similar benefits from neuromorphic computing.

I hope that as these advances are made, Predictive Coding and other biologically inspired frameworks of neural computation will shine in the spotlight as a path forward.

References

[1] N. Akhtar and A. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 2018.

[2] W. R. Ashby. Principles of the self-organizing system. In H. Von Foerster and Jr. G. W. Zopf, editors, *Principles of Self-Organization: Transactions of the University of Illinois Symposium*, pages 255–278, 1962.

[3] P. Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, July 2012.

[4] R. Bogacz. A tutorial on the free-energy framework for modelling perception and learning. *Journal of Mathematical Psychology*, 76:198–211, 2017.

[5] R. Bogacz and K. Gurney. The basal ganglia and cortex implement optimal decision making between alternative actions. *Neural Computation*, 19:442–477, 2007.

[6] R. Bogacz, E. M. Moraud, A. Abdi, P. J. Magill, and J. Baufreton. Properties of neurons in external globus pallidus can support optimal action selection. *PLoS Computational Biology*, 12(7), 2016.

[7] A. Bubic, D. Y. von Cramon, and R. I. Schubotz. Prediction, cognition, and the brain. *Frontiers in Human Neuroscience*, 4:1–15, 2010.

[8] A. Clark. Whatever next? predictive brains, situated agents, and the future of cognitive science. *Behavioral and Brain Sciences*, 36:181–204, 2013.

[9] F. Crick. The recent excitement about neural networks. *Nature*, 337:129–132, 1989.

[10] R. Desimone and J. Duncan. Neural mechanisms of selective visual attention. *Annual Review of Neuroscience*, 18:193–222, 1995.

[11] D. Eriksson, T. Wunderle, and K. Schmidt. Visual cortex combines a stimulus and an error-like signal with a proportion that is dependent on time, space, and stimulus contrast. *Frontiers in Systems Neuroscience*, 6(26), 2012.

[12] B. A. Richards et al. A deep learning framework for neuroscience. *Nature Neuroscience*, 22(11):1761–1770, 2019.

[13] K. Friston. A theory of cortical responses. *Philosophical Transactions Of The Royal Society B*, 360(1456):815–836, 2005.

[14] K. Friston. The free-energy principle: a rough guide to the brain? *Trends In Cognitive Sciences*, 13(7):293–301, 2009.

[15] K. Friston and S. Kiebel. Predictive coding under the free-energy principle. *Philosophical Transactions Of The Royal Society B*, 364(1521):1211–1221, 2009.

[16] A. Golatkar, A. Achille, and S. Soatto. Time matters in regularizing deep networks: Weight decay and data augmentation affect early learning dynamics, matter little near convergence. In *Neural Information Processing Systems*, December 2019.

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, fourth edition, 1996.

[18] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Neural Information Processing Systems*, December 2014.

[19] S. Grossberg. Competitive learning: from interactive activation to adaptive resonance. *Cognitive Science*, 11:23–63, 1987.

[20] J. Guerguiev, T. P. Lillicrap, and B. A. Richards. Towards deep learning with segregated dendrites. *eLife*, 6(e22901), 2017.

[21] K. Han, H. Wen, Y. Zhang, D. Fu, E. Culurciello, and Z. Liu. Deep predictive coding network with local recurrent processing for object recognition. In *Neural Information Processing Systems*, December 2018.

[22] G. F. Harpur and R. Prager. Development of low entropy coding in a recurrent network. *Network: Computation in Neural Systems*, 7:277–284, 1996.

[23] G. E. Hinton. To recognize shapes, first learn to generate images. *Progress in Brain Research*, 165:535–547, 2007.

[24] Y. Huang and R. P. Rao. Predictive coding. *WIREs Cognitive Science*, 2(5):580–593, 2011.

[25] J.M. Hupe, A. C. James, B. R. Payne, S. G. Lomber, P. Girard, and J. Bullier. Cortical feedback improves discrimination between figure and background by V1, V2 and V3 neurons. *Nature*, 394(6695):784–787, 1998.

[26] R. R. Johnson and A. Burkhalter. A polysynaptic feedback circuit in rat visual cortex. *The Journal of Neuroscience*, 17(18):7129–7140, 1997.

[27] E. R. Kandel, J. H. Schwartz, T. M. Jessel, S. A. Siegelbaum, and A. J. Hudspeth. *Principles of Neural Science*. McGraw-Hill Education, New York City, New York, fifth edition, 2012.

[28] D. P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, May 2015.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems*, December 2012.

[30] S. Kullback. *Information Theory and Statistics*. John Wiley and Sons Inc., Hoboken New Jersey, 1959.

[31] S. Kullback. Letters to the editor. *The American Statistician*, 41(4):338–341, 1987.

[32] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[33] M. E. Larkum, W. Senn, and H.-R. Luscher. Top-down dendritic input increases the gain of layer 5. *Cerebral Cortex*, 14(10):1059–1070, 2004.

[34] S. Laughlin. Coding efficiency and visual processing. In M. Pointon C. Blakemore, K. Addler, editor, *Vision: Coding and Efficiency*, chapter 2, pages 25–31. Cambridge University Press, Cambridge, UK, 1990.

[35] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. *Shape, Contour and Grouping in Computer Vision*. Springer-Verlag, Berlin Heidelberg, first edition, 1999.

[36] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 556–562, 2001.

[37] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, 7:13276 EP, 2016.

[38] W. Lotter, G. Kreiman, and D. Cox. Deep predictive coding networks for video prediction and unsupervised learning. In *International Conference on Learning Representations*, April 2017.

[39] J. Makhoul. Linear prediction: A tutorial review. *Proceedings of the IEEE*, 63:561–580, 1975.

[40] K. Mizuseki and G. Buszaki. Preconfigured, skewed distribution of firing rates in the hippocampus and entorhinal cortex. *Cell Reports*, 4(5), 2013.

[41] R. K. Nagle and E. B. Saff. *Fundamentals of Differential Equations*. The Benjamin-Cummings Publishing Company, Inc., Redwood City, CA, second edition, 1989.

[42] B. A. Olshausen and D. J. Field. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research*, 37(23):3311–3325, 1996.

[43] J. Orchard, W. Sun, and N. Liu. Why aren't all predictive coding networks generative? In *Neural Information Processing Systems*, December 2019.

[44] D. O'Shaughnessy. Linear predictive coding. *IEEE Potentials*, 7:29–32, 1988.

[45] S. P. Peron and F. Gabbiani. Role of spike-frequency adaptation in shaping neuronal response to dynamic stimuli. *Biological Cybernetics*, 100(6), 2009.

[46] W. A. Phillips. On the cognitive functions of intracellular mechanisms for contextual amplification. *Brain and Cognition*, 112:39–53, 2017.

[47] R. P. Rao and D. H. Ballard. Predictive coding in the visual cortex: A functional interpretation of some extra-classical receptive-field effects. *Nature Neuroscience*, 2(1):79–87, 1999.

[48] L. Reddy, N. Tsuchiya, and T. Serre. Reading the mind's eye: decoding category information during mental imagery. *NeuroImage*, 50(2):818–825, 2011.

[49] R. D. Reed and R. J. Marks II. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. A Bradford Book, Cambridge, MA, 1999.

[50] J. H. Reynolds, L. Chelazzi, and R. Desimone. Competitive mechanisms subserve attention in macaque areas V2 and V4. *Journal of Neuroscience*, 19(5):1736–1753, 1999.

[51] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[52] Z. Shao and A. Burkhalter. Different balance of excitation and inhibition in forward and feedback circuits of rat visual cortex. *The Journal of Neuroscience*, 16(22):7353–7365, 1996.

[53] Y. Singer, Y. Teramoto, B. D. B. Willmore, J. W. H. Schnupp, A. J. King, and N. S. Harper. Sensory cortex is optimized for prediction of future input. *eLife*, 7:e31557, 2018.

[54] L. L. Solbakken and S. Junge. Online parts-based feature discovery using competitive activation neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1466–1473, July 2011.

[55] M. W. Spratling. Predictive coding as a model of biased competition in visual selective attention. *Vision Research*, 48(12):1391–1408, 2008.

[56] M. W. Spratling. Reconciling predictive coding and biased competition models of cortical function. *Frontiers in Computational Neuroscience*, 2(4):1–8, 2008.

[57] M. W. Spratling. Predictive coding as a model of response properties in cortical area V1. *The Journal of Neuroscience*, 30(9):3531–3543, 2010.

[58] M. W. Spratling. Predictive coding. In D. Jaeger and R. Jung, editors, *Encyclopedia of Computational Neuroscience*, pages 1–5. Springer, New York, NY, 2014.

[59] M. W. Spratling. A review of predictive coding algorithms. *Brain and Cognition*, 112:92–97, 2017.

[60] M. W. Spratling and M. H. Johnson. Dendritic inhibition enhances neural coding properties. *Cerebral Cortex*, 11(12):1144–1149, 2001.

[61] M. W. Spratling and M. H. Johnson. Preintegration lateral inhibition enhances unsupervised learning. *Neural Computation*, 14(9):2157–2179, 2002.

[62] M. W. Spratling, K. De Meyer, and R. Kompass. Unsupervised learning of overlapping image components using divisive input modulation. *Computational Intelligence and Neuroscience*, 2009(381457):1–19, 2009.

[63] M. V. Srinivasan, S. B. Laughlin, and A. Dubs. Predictive coding: A fresh view of inhibition in the retina. *Proceedings of the Royal Society of London, Series B Biological Sciences*, 216:427–459, 1982.

[64] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[65] S. Strogatz. *Nonlinear Dynamics and Chaos*. Westview Press, Cambridge, 1994.

[66] W. Sun and J. Orchard. A predictive coding network that is both discriminative and generative. *Neural Computation*, 32(10), 2020.

[67] S. V. Vaseghi. *Advanced Digital Signal Processing and Noise Reduction.* John Wiley and Sons Inc., Hoboken, New Jersey, second edition, 2000.

[68] J. C. R. Whittington and R. Bogacz. An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity. *Neural Computation*, 29(5):1229–1262, 2017.

APPENDICES

Appendix A

Dr. Orchard's Proof of Theorem 1

Theorem 1: Given a matrix of r linearly-independent m-vectors,

$$X = [X_1|\cdots|X_r] \in \mathbb{R}^{m\times r}$$

and a corresponding matrix of n-vectors,

$$Y = [Y_1|\cdots|Y_r] \in \mathbb{R}^{n\times r}$$

with $r \leq n < m$, there is an $n \times m$ matrix,

$$A = \left[\begin{array}{c} A_1 \\ \hline \vdots \\ \hline A_n \end{array}\right]$$

such that the minimum 2-norm solution x^* to $Ax = Y_i$ is $x^* = X_i$. Moreover, the jth row of A is the minimum 2-norm solution of $aX = Y$ for $a \in \mathbb{R}^{1\times m}$.

Proof. Consider the system $X^T A^T = Y^T$, with r equations and m unknowns. Let the columns of X be linearly independent. Let y_j be the jth row of Y. Then $X^T A_j^T = y_j^T$. This system is under-determined, since $r < m$. Thus, there are infinitely many solutions. However, we can seek the minimum-norm solution for A_j^T using a technique called singular value decomposition, or SVD [17].

Let $U\Sigma V^T = X^T$. Here, U is an $r \times r$ orthogonal matrix, which is a square matrix whose rows and columns are orthonormal, meaning that any pair of rows or columns dot product to zero and they are all unit vectors. V^T is $r \times m$ with orthonormal rows. Σ is a diagonal $r \times r$ matrix containing the r non-zero singular values of the decomposition.

The minimum 2-norm solution of $X^T A_j^T = y_j^T$ is

$$A_j^T = V\Sigma^{-1}U^T y_j^T \,.$$

We can construct all n columns of A^T using $A^T = V\Sigma^{-1}U^T Y^T$.

Our goal is to show that X is a solution of $AX = Y$. Substituting the above expression

for A, followed by the SVD for X^T, we get

$$\begin{aligned} AX &= YU\Sigma^{-1}V^TX \\ &= YU\Sigma^{-1}V^T\left(V\Sigma U^T\right) \\ &= YUU^T \quad \text{since } V^TV = I \text{ and } \Sigma^{-1}\Sigma = I \\ &= Y \quad \text{since } UU^T = I \end{aligned}$$

Thus, X is a solution of $AX = Y$.

Now, we want to show that each column of X is the minimum 2-norm solution. Consider the ith column of X, and suppose we find a different solution, $X_i + \tilde{x}$, where $\tilde{x} \neq 0$. Then,

$$\begin{aligned} A\left(X_i + \tilde{x}\right) &= Y_i \\ AX_i + A\tilde{x} &= Y_i \\ Y_i + A\tilde{x} &= Y_i \\ A\tilde{x} &= 0 \end{aligned}$$

Thus, $\tilde{x} \in \text{null}(A)$, which tells us that $V^T\tilde{x} = 0$. But $X^T = U\Sigma V^T$, so $\tilde{x} \in \text{null}(X^T)$ too. Thus, $X_i \perp \tilde{x}$, meaning they are orthogonal and dot product to 0.

Consider $\|X_i + \tilde{x}\|$. Since $X_i \perp \tilde{x}$, they form the non-hypotenuse edges of a right angle

triangle, and so we can use Pythagoras to deduce the following:

$$\begin{aligned} \|X_i + \tilde{x}\|^2 &= \|X_i\|^2 + \|\tilde{x}\|^2 \\ \|X_i + \tilde{x}\|^2 &> \|X_i\|^2 \quad \text{since } \tilde{x} \neq 0 \\ \implies \|X_i + \tilde{x}\| &> \|X_i\| \end{aligned}$$

Therefore, X_i is the minimum 2-norm solution to $Ax = Y_i$. □