

Introduction to Predictive Coding Networks for Machine Learning

May 29, 2025

Abstract

Predictive coding networks (PCNs) constitute a biologically inspired framework for understanding hierarchical computation in the brain, and offer an alternative to traditional feedforward neural networks in ML. This note serves as a quick, onboarding introduction to PCNs for machine learning practitioners. We cover the foundational network architecture, inference and learning update rules, and algorithmic implementation. A concrete image-classification task (CIFAR-10) is provided as a benchmark-smashing application, together with an accompanying Python notebook containing the PyTorch implementation.

Contents

1	Introduction	2
2	Network Architecture	3
3	Inference and Learning Rules	4
3.1	Latent state update rule (inference)	4
3.2	Weight update rule (learning)	5
3.3	Locality of the updates	6
3.4	Motivations beyond biological plausibility	6
4	Base Algorithms	7
4.1	Unsupervised learning in PCNs	7
4.2	Supervised learning extension	8
5	Application: Supervised Learning on CIFAR-10	10
5.1	Model architecture	10
5.2	Training procedure	11
5.3	Testing	12
5.4	PyTorch implementation	12
5.4.1	The <code>PCNLayer</code> class	13
5.4.2	The <code>PredictiveCodingNetwork</code> class	15
5.4.3	Training loop	16
5.5	Results and observations	18
	References	20

1 Introduction

The goal of this document is to present a first introduction to predictive coding networks from both conceptual and algorithmic standpoints, in the context of machine learning. We focus on detailing the structure of PCNs, deriving their inference and learning rules, and demonstrating their usefulness in an experiment. This section provides a brief overview to set the stage.

Predictive coding is a foundational theory in neuroscience proposing that the brain is fundamentally a prediction machine, constantly attempting to anticipate incoming sensory inputs and updating internal representations to minimize the difference between expectations and actual observations. This concept forms the basis of the predictive coding network (PCN), a class of hierarchical generative models designed to mirror these principles in artificial systems.

Already in 1867, Helmholtz proposed that perception is an unconscious inferential process where the brain predicts how planned actions will affect sensory inputs [12]. In the mid-20th century, Barlow’s efficient coding hypothesis posited that the brain economizes neural representation by removing predictable redundancies in sensory signals [3], foreshadowing a later focus on unexpected or surprising sensory events. Gregory further argued that perception is a constructive, hypothesis-driven endeavor, with visual illusions highlighting how top-down expectations shape perception [10]. By the 1990s, theoretical models explicitly invoked hierarchical prediction: Mumford proposed a cortical architecture in which higher-level areas send predictions to lower-level areas, with only residual errors fed forward [21]. Such concepts set the stage for the predictive coding model of visual cortex by Rao and Ballard [23], which formalized perception as a hierarchical interplay of top-down predictions and bottom-up error signals.

Originally developed to explain extra-classical receptive field effects in visual cortex [23], predictive coding has since been generalized into a unified framework for cortical processing through the *free-energy principle* [8, 9]. This principle casts perception and action as inference problems, where organisms minimize a variational bound on surprise or prediction error.

Neurophysiological plausibility of predictive coding has been further explored in works such as [28], which unify predictive coding with biased competition models of attention, and [4], which describe potential microcircuit implementations in cortical hierarchies. Keller and Mrsic-Flogel [13] offer evidence that predictive processing is a canonical computation across the cortex, with supportive anatomical and functional data reviewed by Shipp [26]. Further empirical neuroscience evidence is presented in [30, 5].

From a computational modeling perspective, predictive coding networks provide an alternative to traditional feedforward models trained via backpropagation. Notably, Whittington and Bogacz [31] demonstrated that predictive coding networks can approximate backpropagation in multilayer neural networks using only local updates; see also Millidge et al. [19]. Their implementation in spiking neural networks was reviewed in [22].

Predictive coding has also inspired innovations in unsupervised and self-supervised learning. For example, Schmidhuber [25] proposed the predictability minimization principle echoing the redundancy reduction goal of predictive coding. Lotter et al. [15] introduced deep predictive coding networks for video frame prediction, offering state-of-the-art performance using unsupervised objectives. Moreover, [11] developed a “world model”—a network that learns a latent predictive representation of an agent’s environment—showing how hierarchical prediction and the minimization of surprise can facilitate efficient learning and planning.

A broader theoretical and empirical review of predictive coding in both neuroscience and artificial intelligence is provided in [18], which also outlines future research directions.

The references cited here are just a bite-sized sample of the enormous literature on the subject. The interested reader is advised to look into the bibliographies in those for more coverage, and to search the arXiv for the latest developments. There are also many excellent blog posts on the subject, such as [17, 1].

2 Network Architecture

Model. A PCN consists of $L \geq 1$ layers of latent variables $\mathbf{x}^{(l)} \in \mathbb{R}^{d_l}$, $1 \leq l \leq L$, and an input layer of variables $\mathbf{x}^{(0)} \in \mathbb{R}^{d_0}$. Each layer attempts to predict the state of the layer **below**, so the architecture includes the following **top-down** elements for $0 \leq l \leq L - 1$:

- Weights $\mathbf{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l+1}}$ from layer $l + 1$ to layer l

- Preactivations

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l+1)} \in \mathbb{R}^{d_l}$$

- Predictions

$$\hat{\mathbf{x}}^{(l)} = f^{(l)}(\mathbf{a}^{(l)}) \in \mathbb{R}^{d_l}$$

where $f^{(l)}$ is, often a nonlinear, scalar function applied elementwise

- Prediction errors

$$\boldsymbol{\epsilon}^{(l)} = \mathbf{x}^{(l)} - \hat{\mathbf{x}}^{(l)} \in \mathbb{R}^{d_l}$$

The loss function subject to minimization is the total square prediction error, or **energy**:

$$\mathcal{L} = \frac{1}{2} \sum_{l=0}^{L-1} \|\boldsymbol{\epsilon}^{(l)}\|^2.$$

The network can be viewed as a directed acyclic graph; see Figure 1. The “hanging” root nodes without incoming edges are the input and latent variables; the leaves (end nodes) are the prediction errors. Observe that the generative hierarchy flows from top to bottom, *towards* the input; more on this will be discussed shortly.

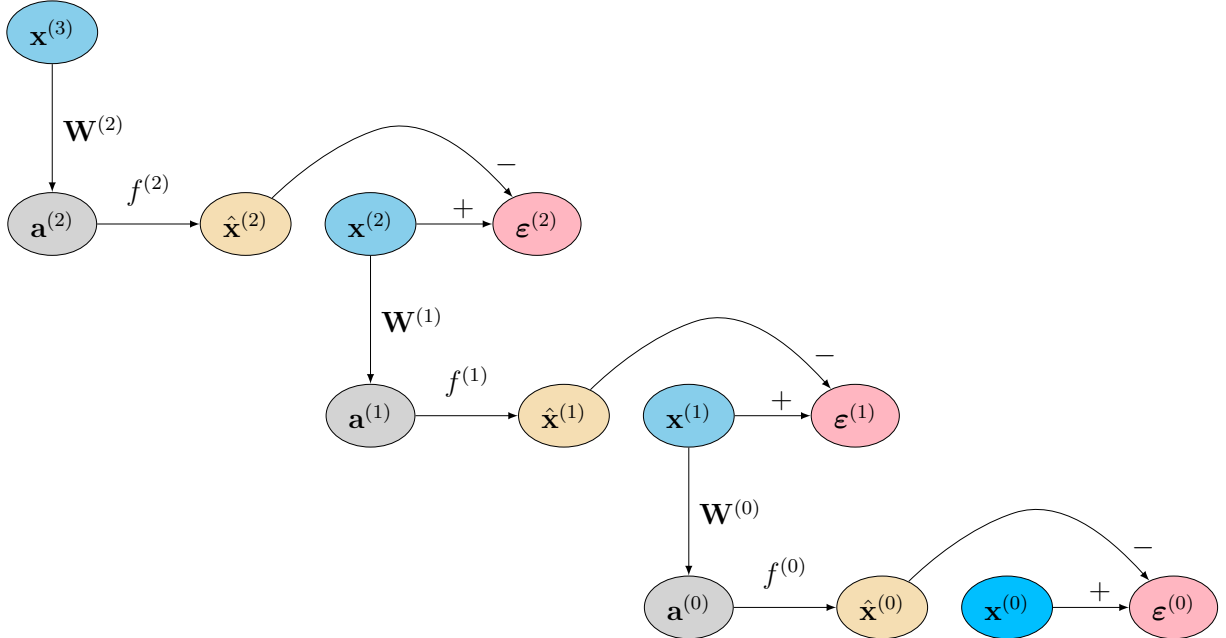


Figure 1: A graph representation of a PCN with three latent layers.

Alternating minimization procedure. The generative weights $\mathbf{W} = (\mathbf{W}^{(0)}, \dots, \mathbf{W}^{(L-1)})$ define the overall shape of the predictive coding energy landscape. For fixed weights and input, the **inference** process seeks a configuration \mathbf{x}^* of latent values $\mathbf{x} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(L)})$ that minimizes the energy:

$$\mathbf{x}^* = \arg \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}; \mathbf{W}, \mathbf{x}^{(0)}) .$$

Optimization then switches to **learning**, which seeks to perturb the energy landscape by adjusting the weights gently to $\mathbf{W}' = \mathbf{W} + \delta \mathbf{W}$ (e.g., taking one gradient step with a small learning rate) so as to further reduce the energy at the inferred configuration \mathbf{x}^* :

$$\mathcal{L}(\mathbf{x}^*; \mathbf{W}', \mathbf{x}^{(0)}) < \mathcal{L}(\mathbf{x}^*; \mathbf{W}, \mathbf{x}^{(0)}) .$$

In this way, inference performs descent within a fixed landscape, while learning deforms the landscape to better accommodate future inference. This separation of roles gives predictive coding networks their characteristic two-timescale dynamics: fast inference, slow learning.

In practice, the procedure repeats for a new input $\mathbf{x}^{(0)}$ and reinitialized latent configuration \mathbf{x} . Such matters are deferred to the last section where an actual application will be discussed together with the training details.

Convergence. Several works have established, formally and experimentally, that the alternating optimization procedure in predictive coding networks—consisting of inference steps over latent variables followed by learning updates of the generative weights—converges to a local minimum of the loss under sufficient assumptions [31, 27, 7, 16, 2, 24]. That said, what matters to practitioners is performance in task-specific settings rather than just guarantees of local convergence under idealized conditions. On that note, the application in the last section serves as one example where fast convergence and excellent generalization do occur.

Generative hierarchy. The network is built in stacked layers of latent variables. The idea is that each layer represents the data at a different level of abstraction, from raw sensory inputs up through progressively higher-order features. Predictions flow top-down (higher layers predicting lower-layer activity) and errors flow bottom-up. Similar processes are believed to exist in the brain’s multi-level sensory hierarchies. Rather than just learning a discriminative mapping from inputs to labels, the PCN explicitly encodes a generative model of how lower-level activity could be produced from higher-level causes, so the network can attempt to reconstruct what the sensory data should look like given its latent beliefs.

A word about hybrid predictive coding. In the algorithms and implementation presented in this document, the initial values of the latent variables are random—both literally and figuratively. Recently, an interesting hybrid predictive coding model has been proposed, where the latents $\mathbf{x}^{(l)}$ are initialized to the values $\boldsymbol{\xi}^{(l)}$ predicted by another network [29]. The predictions of this second network flow in the bottom-up direction, opposite to the PCN hierarchy, via “amortized” functions $\mathbf{g}^{(l)} : \mathbb{R}^{d_{l-1}} \rightarrow \mathbb{R}^{d_l}$ in a feedforward fashion: $\boldsymbol{\xi}^{(l)} = \mathbf{g}^{(l)}(\boldsymbol{\xi}^{(l-1)})$, $1 \leq l \leq L$, starting from the input $\boldsymbol{\xi}^{(0)} = \mathbf{x}^{(0)}$. Such a construction reflects the adjustment of the network’s posterior beliefs upon receiving sensory input, before the refining inference process of the vanilla PCN begins. The rest of this note does not involve the hybrid model.

3 Inference and Learning Rules

3.1 Latent state update rule (inference)

We aim to update each latent variable $\mathbf{x}^{(l)}$ ($1 \leq l \leq L$) via gradient descent on \mathcal{L} .

Case $1 \leq l < L$. The variable $\mathbf{x}^{(l)}$ appears in two places in the loss: in $\boldsymbol{\varepsilon}^{(l)} = \mathbf{x}^{(l)} - \hat{\mathbf{x}}^{(l)}$ explicitly and in $\boldsymbol{\varepsilon}^{(l-1)} = \mathbf{x}^{(l-1)} - \hat{\mathbf{x}}^{(l-1)}$ through $\hat{\mathbf{x}}^{(l-1)} = f^{(l-1)}(\mathbf{W}^{(l-1)}\mathbf{x}^{(l)})$. Hence,

$$\frac{\partial \mathcal{L}}{\partial x_i^{(l)}} = \sum_j \varepsilon_j^{(l)} \frac{\partial \varepsilon_j^{(l)}}{\partial x_i^{(l)}} + \sum_j \varepsilon_j^{(l-1)} \frac{\partial \varepsilon_j^{(l-1)}}{\partial x_i^{(l)}} = \varepsilon_i^{(l)} - \sum_j \varepsilon_j^{(l-1)} \frac{\partial \hat{x}_j^{(l-1)}}{\partial x_i^{(l)}}$$

where

$$\frac{\partial \hat{x}_j^{(l-1)}}{\partial x_i^{(l)}} = \frac{\partial}{\partial x_i^{(l)}} (f^{(l-1)}(a_j^{(l-1)})) = f^{(l-1)'}(a_j^{(l-1)}) W_{ji}^{(l-1)}.$$

Hence, the gradient with respect to $\mathbf{x}^{(l)}$ is

$$\nabla_{\mathbf{x}^{(l)}} \mathcal{L} = \boldsymbol{\varepsilon}^{(l)} - \mathbf{W}^{(l-1)\top} \left(f^{(l-1)'}(\mathbf{a}^{(l-1)}) \odot \boldsymbol{\varepsilon}^{(l-1)} \right) \quad (1 \leq l < L)$$

where \odot denotes the elementwise (Hadamard) product, $f^{(l-1)'}$ is the derivative of $f^{(l-1)}$, and the convention used for stacking the entries is that the vectors $\mathbf{x}^{(l)}$ and $\nabla_{\mathbf{x}^{(l)}} \mathcal{L}$ have the same shape.

Case $l = L$. Notice that $\mathbf{x}^{(L)}$ only appears inside $\boldsymbol{\varepsilon}^{(L-1)}$ in the expression of \mathcal{L} . Hence,

$$\frac{\partial \mathcal{L}}{\partial x_i^{(L)}} = \sum_j \varepsilon_j^{(L-1)} \frac{\partial \varepsilon_j^{(L-1)}}{\partial x_i^{(L)}} = - \sum_j \varepsilon_j^{(L-1)} \frac{\partial \hat{x}_j^{(L-1)}}{\partial x_i^{(L)}} = - \sum_j \varepsilon_j^{(L-1)} f^{(L-1)'}(a_j^{(L-1)}) W_{ji}^{(L-1)}.$$

In vector form,

$$\nabla_{\mathbf{x}^{(L)}} \mathcal{L} = -\mathbf{W}^{(L-1)\top} \left(f^{(L-1)'}(\mathbf{a}^{(L-1)}) \odot \boldsymbol{\varepsilon}^{(L-1)} \right)$$

which is similar to $1 \leq l < L$ except for the missing first term: there is no prediction error for the top layer.

Introducing the convenient constant

$$\boldsymbol{\varepsilon}^{(L)} = \mathbf{0}$$

the **inference update** rule for the gradient descent algorithm becomes compactly

$$\boxed{\mathbf{x}^{(l)} \leftarrow \mathbf{x}^{(l)} - \eta_{\text{infer}} \left(\boldsymbol{\varepsilon}^{(l)} - \mathbf{W}^{(l-1)\top} \left(f^{(l-1)'}(\mathbf{a}^{(l-1)}) \odot \boldsymbol{\varepsilon}^{(l-1)} \right) \right)} \quad (1 \leq l \leq L)$$

where $\eta_{\text{infer}} > 0$ is an inference rate of choice.

During inference, all prediction errors and feedback terms are computed first using the current network state, and only then are the latent variables $\mathbf{x}^{(l)}$ updated. This ensures that each update step is based on a consistent energy landscape and avoids using partially updated states within the same iteration. Conceptually, this corresponds to a synchronous update scheme where all neurons compute their next state based on the same network snapshot.

3.2 Weight update rule (learning)

Each weight matrix $\mathbf{W}^{(l)}$ is responsible for predicting $\mathbf{x}^{(l)}$ from $\mathbf{x}^{(l+1)}$, and appears only in $\boldsymbol{\varepsilon}^{(l)} = \mathbf{x}^{(l)} - f^{(l)}(\mathbf{W}^{(l)}\mathbf{x}^{(l+1)})$. To minimize the loss, we compute

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)}} = \sum_k \varepsilon_k^{(l)} \frac{\partial \varepsilon_k^{(l)}}{\partial W_{ij}^{(l)}} = - \sum_k \varepsilon_k^{(l)} \frac{\partial}{\partial W_{ij}^{(l)}} (f^{(l)}(a_k^{(l)}))$$

which yields

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)}} = - \sum_k \varepsilon_k^{(l)} f^{(l)'}(a_k^{(l)}) \delta_{ik} x_j^{(l+1)} = - \varepsilon_i^{(l)} f^{(l)'}(a_i^{(l)}) x_j^{(l+1)}.$$

As before, using the convention that the matrices $\mathbf{W}^{(l)}$ and $\nabla_{\mathbf{W}^{(l)}} \mathcal{L}$ have the same shape, we arrive at the gradient

$$\nabla_{\mathbf{W}^{(l)}} \mathcal{L} = - \left(f^{(l)'}(\mathbf{a}^{(l)}) \odot \boldsymbol{\varepsilon}^{(l)} \right) \mathbf{x}^{(l+1)\top}.$$

In particular, the **learning update** rule via gradient descent is

$$\boxed{\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} + \eta_{\text{learn}} \left(f^{(l)'}(\mathbf{a}^{(l)}) \odot \boldsymbol{\varepsilon}^{(l)} \right) \mathbf{x}^{(l+1)\top}} \quad (0 \leq l < L)$$

with a learning rate of $\eta_{\text{learn}} > 0$.

Notice that the quantities

$$\mathbf{h}^{(l)} = f^{(l)'}(\mathbf{a}^{(l)}) \odot \boldsymbol{\varepsilon}^{(l)} \quad (0 \leq l < L)$$

are central, as they appear in the expressions of the gradients with respect to both the latents and the weights. They could be called **gain-modulated errors**; see the subsection below.

3.3 Locality of the updates

One of the original motivations behind predictive coding is its potential biological plausibility: that the brain could implement something akin to deep hierarchical learning using local computations. Locality typically refers to whether a computation depends only on information from a given layer and its immediate neighbors. This concept is important both for computational efficiency and biological plausibility.

The **learning update** for the weight matrix $\mathbf{W}^{(l)}$ is **local in a strong sense**: it depends only on the local activity $\mathbf{x}^{(l+1)}$ of layer $l + 1$ (presynaptic) and the local prediction error $\boldsymbol{\varepsilon}^{(l)}$ at layer l (postsynaptic), making it compatible with Hebbian-like plasticity mechanisms often summarized as “neurons that fire together, wire together.”

In contrast, while the **inference update** for a latent variable $\mathbf{x}^{(l)}$ depends only on adjacent layers, it requires access to the error signal $\boldsymbol{\varepsilon}^{(l-1)}$ broadcast from the lower layer, modulated by the top-down weights in $\mathbf{W}^{(l-1)}$. Thus, inference updates are layer-local but not neuron-local, since a neuron’s update depends on a weighted sum of errors from other neurons in the layer below.

Biologically, the learning updates $\delta W_{ij}^{(l)} = -\eta_{\text{learn}} \varepsilon_i^{(l)} f^{(l)'}(a_i^{(l)}) x_j^{(l+1)}$ are thought to be more plausible than inference updates. Here, $\varepsilon_i^{(l)}$ is the prediction error at the postsynaptic neuron i , $a_i^{(l)} = \sum_m W_{im}^{(l)} x_m^{(l+1)}$ is its preactivation, and $x_j^{(l+1)}$ is the activity of the presynaptic neuron j . The preactivation represents the total synaptic input to neuron i —essentially its membrane potential or driving current. In both biological and artificial neurons, this summation is performed naturally as part of neural activation. The neuron does not need to access other neurons’ states; it simply integrates the inputs it receives via its dendrites. Thus, $f^{(l)'}(a_i^{(l)})$ can be interpreted as a local gain or nonlinearity applied to the neuron’s own internal state. This makes the full weight update neuron-local in the strong sense: it depends only on information accessible to the synapse between neurons $j \rightarrow i$, including presynaptic activity, postsynaptic error, and internal quantities of the postsynaptic neuron.

3.4 Motivations beyond biological plausibility

While predictive coding networks (PCNs) are often motivated by neuroscience and cortical modeling, their architectural design makes them attractive for future machine learning systems on emerging (e.g., neuromorphic) hardware.

Locality and parallelism. PCNs rely on local computations: each weight update depends only on the activity and prediction error of adjacent neurons. This makes them well suited to distributed and parallel computing, in contrast to global gradient backpropagation.

Separation of algorithmic and physical synchrony. Inference in PCNs is described here as a synchronous algorithm: each update step operates on a fixed snapshot of the network state. This ensures convergence and simplifies analysis. In principle, the underlying computations could be implemented asynchronously in hardware, without needing a global clock or centralized scheduling.

Decentralized control. Unlike backpropagation, which requires tightly coordinated forward and backward passes, PCNs do not rely on a global gradient tape or synchronized dataflow. This allows them to operate on hardware with minimal coordination or shared memory.

Energy-efficient, adaptive inference. PCNs support variable-length inference: predictable inputs can settle in fewer steps, while uncertain or surprising stimuli drive deeper inference. This adaptivity enables anytime computation, useful in energy-constrained settings such as embedded devices or robotics.

Architectural versatility. PCNs extend naturally to convolutional, recurrent, and graph-based structures by redefining local prediction and feedback pathways.

In short, predictive coding offers a biologically inspired view of computation whose design pattern aligns with the demands of emerging hardware. These properties make PCNs a high-value target for research in scalable, low-power, and distributed learning systems [6, 19].

4 Base Algorithms

4.1 Unsupervised learning in PCNs

This algorithm implements unsupervised learning in a predictive coding network with L layers of latent variables $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(L)}$, with the input variables $\mathbf{x}^{(0)}$ clamped to the input data; recall Figure 1 in Section 1. The latent variables are inferred via iterative updates to minimize the global prediction error energy as discussed earlier, starting from a random initial state. Each inference step uses a consistent snapshot of the network: all prediction errors and gradients are computed before any latent state is updated.

Each layer l receives top-down predictions from layer $l + 1$ through a learned weight matrix $\mathbf{W}^{(l)}$ and nonlinearity $f^{(l)}$. After inference, weights are updated using local Hebbian-like learning rules based on the final prediction errors.

Per-sample training. The base algorithm presented here describes a single inference-learning cycle for one input sample. Training proceeds by repeating this cycle over many samples drawn from a dataset. For each sample, latent variables are inferred and weights are updated once. For mini-batch training discussed later, the inference loop can be run for the samples in the current batch in parallel, and the subsequent weight update(s) can be carried out using the mean gradient over the batch.

Algorithm 1 Unsupervised learning in a predictive coding network

Require: Input $\mathbf{x}^{(0)}$, generative weights $\{\mathbf{W}^{(l)}\}_{l=0}^{L-1}$, activation functions and their derivatives $\{f^{(l)}, f^{(l)'}\}$, number of inference steps T_{infer} , learning rate η_{learn} , inference rate η_{infer}

- 1: **Clamp** $\mathbf{x}^{(0)} \leftarrow$ input data
- 2: **for** layer $l = 1$ to L **do**
- 3: Initialize $\mathbf{x}^{(l)} \leftarrow$ small random values
- 4: **end for**
- 5: $\boldsymbol{\varepsilon}^{(L)} \leftarrow \mathbf{0}$ ▷ Top layer has no prediction error
- 6: **for** step $t = 1$ to T_{infer} **do** ▷ **Inference update loop**
- 7: **for** layer $l = 0$ to $L - 1$ **do** ▷ Store network state snapshot
- 8: $\mathbf{a}^{(l)} \leftarrow \mathbf{W}^{(l)} \mathbf{x}^{(l+1)}$ ▷ Preactivation
- 9: $\hat{\mathbf{x}}^{(l)} \leftarrow f^{(l)}(\mathbf{a}^{(l)})$ ▷ Prediction
- 10: $\boldsymbol{\varepsilon}^{(l)} \leftarrow \mathbf{x}^{(l)} - \hat{\mathbf{x}}^{(l)}$ ▷ Prediction error
- 11: **end for**
- 12: **for** layer $l = 1$ to L **do** ▷ Update latents using snapshot
- 13: $\mathbf{g}_{\mathbf{x}}^{(l)} \leftarrow \boldsymbol{\varepsilon}^{(l)} - \mathbf{W}^{(l-1)\top} (f^{(l-1)' }(\mathbf{a}^{(l-1)}) \odot \boldsymbol{\varepsilon}^{(l-1)})$ ▷ Gradient wrt $\mathbf{x}^{(l)}$
- 14: $\mathbf{x}^{(l)} \leftarrow \mathbf{x}^{(l)} - \eta_{\text{infer}} \mathbf{g}_{\mathbf{x}}^{(l)}$
- 15: **end for**
- 16: **end for**
- 17: **for** layer $l = 0$ to $L - 1$ **do** ▷ **Weight update**
- 18: $\mathbf{a}^{(l)} \leftarrow \mathbf{W}^{(l)} \mathbf{x}^{(l+1)}$
- 19: $\hat{\mathbf{x}}^{(l)} \leftarrow f^{(l)}(\mathbf{a}^{(l)})$
- 20: $\boldsymbol{\varepsilon}^{(l)} \leftarrow \mathbf{x}^{(l)} - \hat{\mathbf{x}}^{(l)}$
- 21: $\mathbf{g}_{\mathbf{W}}^{(l)} \leftarrow -(\boldsymbol{\varepsilon}^{(l)} \odot f^{(l)'}(\mathbf{a}^{(l)})) \mathbf{x}^{(l+1)\top}$ ▷ Gradient wrt $\mathbf{W}^{(l)}$
- 22: $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta_{\text{learn}} \mathbf{g}_{\mathbf{W}}^{(l)}$
- 23: **end for**

Remark. The model can also support anytime inference, where well-predicted inputs converge in fewer steps, and additional inference steps are taken to improve predictions in ambiguous cases. This adaptivity offers potential energy savings in embedded or neuromorphic deployments. The algorithm can be modified in this spirit by choosing a sufficiently large maximum step count T_{infer} , and running the inference loop until either T_{infer} steps have been performed or convergence has been detected. Here convergence means, for instance, that the norm of the latest update (or updates over a longer patience window) across all latent variables falls below a preset threshold. In machine learning terminology, this could be phrased as inference with sample-wise early stopping.

4.2 Supervised learning extension

A minimal modification to apply predictive coding in a supervised setting entails simply clamping the top latent representation $\mathbf{x}^{(L)}$ to a predicted label $\hat{\mathbf{y}} \in \mathbb{R}^{d_{\text{out}}}$, treating it as part of the generative hierarchy. For the sake of variation in this pedagogically oriented note, we introduce a separate readout layer that maps $\mathbf{x}^{(L)} \mapsto \hat{\mathbf{y}}$ linearly:

$$\hat{\mathbf{y}} = \mathbf{W}^{\text{out}} \mathbf{x}^{(L)}$$

where $\mathbf{W}^{\text{out}} \in \mathbb{R}^{d_{\text{out}} \times d_L}$. Given a target label $\mathbf{y} \in \mathbb{R}^{d_{\text{out}}}$, we define a supervised error

$$\boldsymbol{\varepsilon}^{\text{sup}} = \hat{\mathbf{y}} - \mathbf{y} .$$

Figure 2 illustrates these changes relative to Figure 1. The loss function (energy) now becomes

$$\mathcal{L} + \mathcal{L}_{\text{sup}}$$

where $\mathcal{L}_{\text{sup}} = \frac{1}{2} \|\boldsymbol{\epsilon}^{\text{sup}}\|^2$ is the supervised energy.

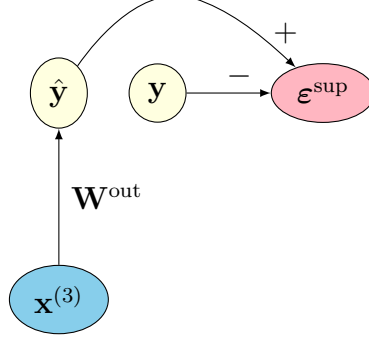


Figure 2: A supervised extension of the PCN with three latent layers in Figure 1 is obtained by stacking a readout layer on top of the highest latent layer, together with a new root node for the target label.

The supervised error is backpropagated into the top latent representation $\mathbf{x}^{(L)}$ during inference. This does not change the update rules of the lower latents $\mathbf{x}^{(l)}$ or the generative weights $\mathbf{W}^{(l)}$, $0 \leq l < L$, at all. Since $\nabla_{\mathbf{x}^{(L)}} \mathcal{L}_{\text{sup}} = \mathbf{W}^{\text{out}\top} \boldsymbol{\epsilon}^{\text{sup}}$, the inference update rule for the top latent is modified to

$$\boxed{\mathbf{x}^{(L)} \leftarrow \mathbf{x}^{(L)} - \eta_{\text{infer}} \left(\mathbf{W}^{\text{out}\top} \boldsymbol{\epsilon}^{\text{sup}} - \mathbf{W}^{(L-1)\top} \left(f^{(L-1)'}(\mathbf{a}^{(L-1)}) \odot \boldsymbol{\epsilon}^{(L-1)} \right) \right)}.$$

Note that this becomes formally the same as in the unsupervised case if we introduce the symbol $\boldsymbol{\epsilon}^{(L)} = \mathbf{W}^{\text{out}\top} \boldsymbol{\epsilon}^{\text{sup}}$ (instead of $\boldsymbol{\epsilon}^{(L)} = \mathbf{0}$), which is how we implement the algorithm.

On the other hand, $\nabla_{\mathbf{W}^{\text{out}}} \mathcal{L}_{\text{sup}} = \boldsymbol{\epsilon}^{\text{sup}} \mathbf{x}^{(L)\top}$. The output weights \mathbf{W}^{out} are thus updated via

$$\boxed{\mathbf{W}^{\text{out}} \leftarrow \mathbf{W}^{\text{out}} - \eta_{\text{learn}} \boldsymbol{\epsilon}^{\text{sup}} \mathbf{x}^{(L)\top}}.$$

The core network structure remains unchanged from the unsupervised case; the only modification is the supervised error signal applied to the top layer.

Algorithm 2 Supervised learning in a predictive coding network

Require: Input $\mathbf{x}^{(0)}$, target label \mathbf{y} , generative weights $\{\mathbf{W}^{(l)}\}_{l=0}^{L-1}$, output weights \mathbf{W}^{out} , activation functions and their derivatives $\{f^{(l)}, f^{(l)'}\}$, inference steps T_{infer} , learning rate η_{learn} , inference rate η_{infer}

- 1: **Clamp** $\mathbf{x}^{(0)} \leftarrow$ input data
- 2: **for** layer $l = 1$ to L **do**
- 3: Initialize $\mathbf{x}^{(l)} \leftarrow$ small random values
- 4: **end for**
- 5: **for** step $t = 1$ to T_{infer} **do** \triangleright **Inference update loop**
- 6: **for** layer $l = 0$ to $L - 1$ **do** \triangleright Store network state snapshot
- 7: $\mathbf{a}^{(l)} \leftarrow \mathbf{W}^{(l)} \mathbf{x}^{(l+1)}$ \triangleright Preactivation
- 8: $\hat{\mathbf{x}}^{(l)} \leftarrow f^{(l)}(\mathbf{a}^{(l)})$ \triangleright Prediction
- 9: $\boldsymbol{\epsilon}^{(l)} \leftarrow \mathbf{x}^{(l)} - \hat{\mathbf{x}}^{(l)}$ \triangleright Prediction error
- 10: **end for**
- 11: $\hat{\mathbf{y}} \leftarrow \mathbf{W}^{\text{out}} \mathbf{x}^{(L)}$ \triangleright Output
- 12: $\boldsymbol{\epsilon}^{\text{sup}} \leftarrow \hat{\mathbf{y}} - \mathbf{y}$ \triangleright Supervised error
- 13: $\boldsymbol{\epsilon}^{(L)} \leftarrow \mathbf{W}^{\text{out}\top} \boldsymbol{\epsilon}^{\text{sup}}$ \triangleright Convenient notation
- 14: **for** layer $l = 1$ to L **do** \triangleright Update latents using snapshot
- 15: $\mathbf{g}_{\mathbf{x}}^{(l)} \leftarrow \boldsymbol{\epsilon}^{(l)} - \mathbf{W}^{(l-1)\top} (f^{(l-1)' }(\mathbf{a}^{(l-1)}) \odot \boldsymbol{\epsilon}^{(l-1)})$
- 16: $\mathbf{x}^{(l)} \leftarrow \mathbf{x}^{(l)} - \eta_{\text{infer}} \mathbf{g}_{\mathbf{x}}^{(l)}$
- 17: **end for**
- 18: **end for**
- 19: **for** layer $l = 0$ to $L - 1$ **do** \triangleright **Weight update**
- 20: $\mathbf{a}^{(l)} \leftarrow \mathbf{W}^{(l)} \mathbf{x}^{(l+1)}$
- 21: $\hat{\mathbf{x}}^{(l)} \leftarrow f^{(l)}(\mathbf{a}^{(l)})$
- 22: $\boldsymbol{\epsilon}^{(l)} \leftarrow \mathbf{x}^{(l)} - \hat{\mathbf{x}}^{(l)}$
- 23: $\mathbf{g}_{\mathbf{W}}^{(l)} \leftarrow -(\boldsymbol{\epsilon}^{(l)} \odot f^{(l)'}(\mathbf{a}^{(l)})) \mathbf{x}^{(l+1)\top}$
- 24: $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta_{\text{learn}} \mathbf{g}_{\mathbf{W}}^{(l)}$
- 25: **end for**
- 26: $\mathbf{g}_{\mathbf{W}}^{\text{out}} \leftarrow \boldsymbol{\epsilon}^{\text{sup}} \mathbf{x}^{(L)\top}$ \triangleright Gradient wrt \mathbf{W}^{out}
- 27: $\mathbf{W}^{\text{out}} \leftarrow \mathbf{W}^{\text{out}} - \eta_{\text{learn}} \mathbf{g}_{\mathbf{W}}^{\text{out}}$

5 Application: Supervised Learning on CIFAR-10

To demonstrate the practical applicability of predictive coding networks, we evaluate a supervised PCN on the CIFAR-10 image classification task. CIFAR-10 consists of 60,000 color images of size $32 \times 32 \times 3$, divided evenly across 10 classes, with 50,000 training and 10,000 test samples [14].

5.1 Model architecture

Each input image is flattened into a vector $\mathbf{x}^{(0)} \in \mathbb{R}^{3072}$, normalized to the range $[0, 1]$. The network is defined with $L = 3$ latent layers, where the topmost latent state $\mathbf{x}^{(L)} \in \mathbb{R}^{10}$ is linearly mapped to an output vector $\hat{\mathbf{y}} \in \mathbb{R}^{10}$ using a readout matrix $\mathbf{W}^{\text{out}} \in \mathbb{R}^{10 \times 10}$. The latent representations are

$$\mathbf{x}^{(1)} \in \mathbb{R}^{1000}, \quad \mathbf{x}^{(2)} \in \mathbb{R}^{500}, \quad \mathbf{x}^{(3)} \in \mathbb{R}^{10}$$

and the top-down generative weights are

$$\mathbf{W}^{(0)} \in \mathbb{R}^{3072 \times 1000}, \quad \mathbf{W}^{(1)} \in \mathbb{R}^{1000 \times 500}, \quad \mathbf{W}^{(2)} \in \mathbb{R}^{500 \times 10}$$

with scalar nonlinearity $f^{(l)} = \text{ReLU} = \max(0, \cdot)$ applied elementwise. No bias terms are used. The network structure is quite arbitrary, following just the battle-tested practice that the layer dimensions interpolate between the input and output in a quick progression.

The total number of trainable parameters in the PCN is

$$3,072 \times 1,000 + 1,000 \times 500 + 500 \times 10 + 10 \times 10 = 3,577,100 .$$

To draw a very loose connection with biology, the PCN comprises roughly 4.6×10^3 “neurons” (3,072 inputs, 1,000 first-order latents, 500 second-order latents, and 10 outputs) connected by 3.6×10^6 “synapses.”

5.2 Training procedure

All weights are initialized using Xavier initialization, once, at the very beginning.

Each input-label pair $(\mathbf{x}^{(0)}, y)$ undergoes the inference and learning cycle, as described in the supervised learning algorithm, with the latent variables $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(L)}$ freshly initialized to small Gaussian noise. The output prediction is computed as $\hat{\mathbf{y}} = \mathbf{W}^{\text{out}} \mathbf{x}^{(L)}$ and the supervised prediction error is given by $\boldsymbol{\varepsilon}^{\text{sup}} = \hat{\mathbf{y}} - \mathbf{y}$ where $\mathbf{y} \in \mathbb{R}^{10}$ is the one-hot encoded target y .

Although the base algorithm was formulated in a per-sample manner, mini-batch training improves computational efficiency and learning stability. We process a batch of B samples $\{(\mathbf{x}_b^{(0)}, \mathbf{y}_b)\}_{b=1}^B$ in parallel. Each sample maintains its own set of latent variables $\mathbf{x}_b^{(l)}$ and errors $\boldsymbol{\varepsilon}_b^{(l)}$, updated independently using the usual inference rule over T iterations. After inference, the weight gradients $\mathbf{g}_b^{(l)} = -(\boldsymbol{\varepsilon}_b^{(l)} \odot f^{(l)'}(\mathbf{a}_b^{(l)}))\mathbf{x}_b^{(l+1)\top}$ and $\mathbf{g}_b^{\text{out}} = \boldsymbol{\varepsilon}_b^{\text{sup}}\mathbf{x}_b^{(L)\top}$ are computed per sample (as in the base algorithm), and averaged over the batch:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta_{\text{learn}} \frac{1}{B} \sum_{b=1}^B \mathbf{g}_b^{(l)}$$

and

$$\mathbf{W}^{\text{out}} \leftarrow \mathbf{W}^{\text{out}} - \eta_{\text{learn}} \frac{1}{B} \sum_{b=1}^B \mathbf{g}_b^{\text{out}}$$

Since the total number of inference updates per batch is $B \times T_{\text{infer}}$, these learning updates are performed B times, thus keeping the ratio of inference updates to learning updates at T_{infer} . For each learning step the same inputs $\mathbf{x}_b^{(0)}$ and inferred latents $\mathbf{x}_b^{(l)}$ are used to recompute the gradients $\mathbf{g}_b^{(l)}$ and $\mathbf{g}_b^{\text{out}}$.

This procedure preserves the sample-wise locality of PCN inference and learning, while leveraging parallel processing on a GPU.

Hyperparameters. For training we set the hyperparameters as follows:

- Batch size: 500 (resulting in 100 train batches, 20 test batches)
- Inference steps per sample: $T_{\text{infer}} = 50$
- Inference rate: $\eta_{\text{infer}} = 0.05$
- Learning steps per batch: $T_{\text{learn}} = 500$ (equals batch size)
- Learning rate: $\eta_{\text{learn}} = 0.005$

Such choices are largely guided by the belief, or inductive bias, that there should be a distinct **separation of timescales**: inference should progress much faster than learning.

5.3 Testing

To test the trained network, the weights $\mathbf{W}^{(l)}$ and \mathbf{W}^{out} are frozen. For each input-label pair $(\mathbf{x}^{(0)}, y)$ from the test set, the latents $\mathbf{x}^{(l)}$ are initialized randomly and the inference loop is executed, exactly as in the base algorithm. Once the latents are optimized, the prediction $\hat{\mathbf{y}}$ is read from the output layer. Observe that, given an input $\mathbf{x}^{(0)}$, the prediction $\hat{\mathbf{y}}$ in practice contains some random noise, since the inference loop starts from randomly initialized latents.

Top-1 and top-3 class-prediction accuracies are used as performance metrics: Let $\text{top}_k(\hat{\mathbf{y}})$ be the set of k indices corresponding to the largest entries of $\hat{\mathbf{y}}$. Then the top- k accuracy is the average of $\mathbf{1}(y \in \text{top}_k(\hat{\mathbf{y}}))$ over the test set. Note that the test accuracies for a well-trained PCN vary slightly from one test round to the next, due to the random initialization.

In the implementation test samples are handled in batches, just like in training.

5.4 PyTorch implementation

It is customary in PyTorch to load data as tensors where the batch dimension comes first. In our case the shape of a batch of samples is then (B, d_0) , corresponding to a matrix $\mathbf{X}^{(0)}$ whose **rows** represent the inputs $\mathbf{x}_b^{(0)} \in \mathbb{R}^{1 \times d_0}$, $1 \leq b \leq B$. To adhere to the custom and to allow for efficient batch handling via vectorization, we revise the supervised learning algorithm to its final form. The changes from earlier essentially amount to adding a batch dimension to all vectors and transposing certain products. As explained above, the learning step is furthermore repeated B times (by default) instead of just once; this counter-balances the increase in the number of inference steps due to batching prior to any weight updates—from T_{infer} per sample in case of no batching ($B = 1$) to $B \times T_{\text{infer}}$ per batch. To make reasoning about dimensions easier, they are listed explicitly on the right side of the algorithm box.

During training we also track the batch-averaged total energy

$$\mathcal{E}_{\text{batch}}(t) = \frac{1}{B} \sum_{b=1}^B \left(\frac{1}{2} \sum_{l=0}^{L-1} \|\boldsymbol{\varepsilon}_b^{(l)}(t)\|^2 + \frac{1}{2} \|\boldsymbol{\varepsilon}_b^{\text{sup}}(t)\|^2 \right)$$

at every inference and learning step t . Here the quadratic terms are the per-sample energies. Since there are 100 batches per epoch, we thus generate 100 “energy trajectories” over the time window $(0, \dots, T_{\text{infer}} + T_{\text{learn}})$ per epoch. Note that for sufficiently small inference and learning rates these trajectories should be strictly decreasing through inference and learning, which can be used as a sanity check that the implementation is correct. In practice, larger rates are required to escape local minima and to facilitate faster convergence.

To support energy-tracking, our implementation reorders a few lines compared to the pseudocode here. Specifically, we compute and log the errors $\boldsymbol{\varepsilon}_b^{(l)}(t)$ and $\boldsymbol{\varepsilon}_b^{\text{sup}}(t)$ pre- and post-update at each step, then reuse those same quantities for the next inference or weight update. This ensures that we record the energy both before any updates (at $t = 0$) and immediately after each subsequent update, without redundant recomputation.

Algorithm 3 Supervised learning in a PCN (vectorized row-batch form)

Require: Input batch $\mathbf{X}^{(0)} \in \mathbb{R}^{B \times d_0}$, target batch $\mathbf{Y} \in \mathbb{R}^{B \times d_{\text{out}}}$, generative weights $\{\mathbf{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l+1}}\}_{l=0}^{L-1}$, output weights $\mathbf{W}^{\text{out}} \in \mathbb{R}^{d_{\text{out}} \times d_L}$, activation functions and their derivatives $\{f^{(l)}, f^{(l)'}\}$, inference steps per sample T_{infer} , learning steps per batch $T_{\text{learn}} = B$, inference rate η_{infer} , learning rate η_{learn}

- 1: **for** layer $l = 1$ to L **do**
- 2: Initialize $\mathbf{X}^{(l)} \leftarrow$ small random values $\triangleright B \times d_l$
- 3: **end for**
- 4: **for** $t = 1$ to T_{infer} **do** \triangleright Inference update loop
- 5: **for** $l = 0$ to $L - 1$ **do**
- 6: $\mathbf{A}^{(l)} \leftarrow \mathbf{X}^{(l+1)} \mathbf{W}^{(l)\top}$ $\triangleright B \times d_l$
- 7: $\hat{\mathbf{X}}^{(l)} \leftarrow f^{(l)}(\mathbf{A}^{(l)})$ $\triangleright B \times d_l$
- 8: $\mathbf{E}^{(l)} \leftarrow \mathbf{X}^{(l)} - \hat{\mathbf{X}}^{(l)}$ $\triangleright B \times d_l$
- 9: $\mathbf{H}^{(l)} = \mathbf{E}^{(l)} \odot f^{(l)'}(\mathbf{A}^{(l)})$ $\triangleright B \times d_l$
- 10: **end for**
- 11: $\hat{\mathbf{Y}} \leftarrow \mathbf{X}^{(L)} \mathbf{W}^{\text{out}\top}$ $\triangleright B \times d_{\text{out}}$
- 12: $\mathbf{E}^{\text{sup}} \leftarrow \hat{\mathbf{Y}} - \mathbf{Y}$ $\triangleright B \times d_{\text{out}}$
- 13: $\mathbf{E}^{(L)} \leftarrow \mathbf{E}^{\text{sup}} \mathbf{W}^{\text{out}}$ $\triangleright B \times d_L$
- 14: **for** $l = 1$ to L **do**
- 15: $\mathbf{G}_{\mathbf{X}}^{(l)} \leftarrow \mathbf{E}^{(l)} - \mathbf{H}^{(l-1)} \mathbf{W}^{(l-1)}$ \triangleright Grads wrt sample latents $\mathbf{x}_b^{(l)}$, batched $\triangleright B \times d_l$
- 16: $\mathbf{X}^{(l)} \leftarrow \mathbf{X}^{(l)} - \eta_{\text{infer}} \mathbf{G}_{\mathbf{X}}^{(l)}$ $\triangleright B \times d_l$
- 17: **end for**
- 18: **end for**
- 19: **for** $t = 1$ to T_{learn} **do** \triangleright Weight update loop
- 20: **for** $l = 0$ to $L - 1$ **do**
- 21: $\mathbf{A}^{(l)} \leftarrow \mathbf{X}^{(l+1)} \mathbf{W}^{(l)\top}$ $\triangleright B \times d_l$
- 22: $\hat{\mathbf{X}}^{(l)} \leftarrow f^{(l)}(\mathbf{A}^{(l)})$ $\triangleright B \times d_l$
- 23: $\mathbf{E}^{(l)} \leftarrow \mathbf{X}^{(l)} - \hat{\mathbf{X}}^{(l)}$ $\triangleright B \times d_l$
- 24: $\mathbf{H}^{(l)} = \mathbf{E}^{(l)} \odot f^{(l)'}(\mathbf{A}^{(l)})$ $\triangleright B \times d_l$
- 25: $\mathbf{G}_{\mathbf{W}}^{(l)} \leftarrow -\frac{1}{B} \mathbf{H}^{(l)\top} \mathbf{X}^{(l+1)}$ \triangleright Batch-averaged grad wrt $\mathbf{W}^{(l)}$ $\triangleright d_l \times d_{l+1}$
- 26: $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta_{\text{learn}} \mathbf{G}_{\mathbf{W}}^{(l)}$ $\triangleright d_l \times d_{l+1}$
- 27: **end for**
- 28: $\hat{\mathbf{Y}} \leftarrow \mathbf{X}^{(L)} \mathbf{W}^{\text{out}\top}$ $\triangleright B \times d_{\text{out}}$
- 29: $\mathbf{E}^{\text{sup}} \leftarrow \hat{\mathbf{Y}} - \mathbf{Y}$ $\triangleright B \times d_{\text{out}}$
- 30: $\mathbf{G}_{\mathbf{W}}^{\text{out}} \leftarrow \frac{1}{B} \mathbf{E}^{\text{sup}\top} \mathbf{X}^{(L)}$ \triangleright Batch-averaged grad wrt \mathbf{W}^{out} $\triangleright d_{\text{out}} \times d_L$
- 31: $\mathbf{W}^{\text{out}} \leftarrow \mathbf{W}^{\text{out}} - \eta_{\text{learn}} \mathbf{G}_{\mathbf{W}}^{\text{out}}$ $\triangleright d_{\text{out}} \times d_L$
- 32: **end for**

Next, key modules of the Python code will be described. Full implementation details are provided in the accompanying Python notebook, available in a [GitHub repository](#) [20].

5.4.1 The PCNLayer class

Each latent layer of the PCN is encapsulated by the `PCNLayer` class, defined as follows:

PCNLayer class

```
class PCNLayer(nn.Module):
    def __init__(self,
                  in_dim,
                  out_dim,
                  activation_fn=torch.relu,
                  activation_deriv=lambda a: (a > 0).float()
                  ):
        super().__init__()
        self.W = nn.Parameter(torch.empty(out_dim, in_dim))
        nn.init.xavier_uniform_(self.W)
        self.activation_fn = activation_fn
        self.activation_deriv = activation_deriv

    def forward(self, x_above):
        with autocast(device_type='cuda'):
            a = x_above @ self.W.T
            x_hat = self.activation_fn(a)
            return x_hat, a
```

The `PCNLayer` class inherits from `torch.nn.Module`, making it a standard building block in the PyTorch ecosystem. Its constructor, `__init__`, initializes the following components:

- **Weights (`self.W`):** The learnable generative weight matrix, corresponding to $\mathbf{W}^{(l)}$ in our model, is initialized as a `torch.nn.Parameter`. This ensures that the weights are recognized by PyTorch’s autograd system and can be updated during the learning phase. The dimensions are `(out_dim, in_dim)`, representing $d_l \times d_{l+1}$, connecting layer $l + 1$ (of `in_dim` neurons) to layer l (of `out_dim` neurons) in a top-down predictive manner. Xavier uniform initialization (`nn.init.xavier_uniform_`) is used to set the initial values of these weights. No biases are used.
- **Activation function (`self.activation_fn`):** This stores the element-wise nonlinear activation function $f^{(l)}$, which defaults to ReLU (`torch.relu`).
- **Activation derivative (`self.activation_deriv`):** This stores the derivative of the activation function, $f^{(l) \prime}$, required for calculating the gain-modulated errors $\mathbf{H}^{(l)}$ during both inference and learning updates. The default is the derivative of ReLU.

The core computation of a layer—generating a prediction for the layer below—is handled by the `forward` method. Given an input `x_above` (representing the state of the layer above, $\mathbf{X}^{(l+1)}$ in batch form), this method performs two main operations:

1. It computes the pre-activation $\mathbf{A}^{(l)} = \mathbf{X}^{(l+1)} \mathbf{W}^{(l)\top}$ (denoted `a` in the code). This corresponds to the matrix multiplication `x_above @ self.W.T`.
2. It then applies the layer’s activation function $f^{(l)}$ to the pre-activations `a` to produce the prediction $\hat{\mathbf{X}}^{(l)}$ (denoted `x_hat` in the code).

The method returns both the prediction `x_hat` and the pre-activation `a`, as both are needed for subsequent error calculations and gradient computations as detailed in the vectorized algorithm (Algorithm 3). By encapsulating these operations in a single module, `PCNLayer` separates the per-layer computations required for both inference (updating latents) and learning (updating weights) in Algorithm 3. The use of `autocast(device_type='cuda')` is for mixed-precision training, which can improve computational efficiency on compatible hardware.

5.4.2 The PredictiveCodingNetwork class

Building upon the PCNLayer, the PredictiveCodingNetwork class organizes the entire network hierarchy and its operations. This class, also inheriting from `torch.nn.Module`, manages the collection of layers, the readout mechanism for supervised tasks, and utility functions for initialization and error computation.

PredictiveCodingNetwork class

```
class PredictiveCodingNetwork(nn.Module):
    def __init__(self,
                  dims,
                  output_dim
                  ):
        super().__init__()
        self.dims = dims
        self.L = len(dims) - 1
        self.layers = nn.ModuleList([
            PCNLayer(in_dim=dims[l+1],
                     out_dim=dims[l])
            for l in range(self.L)
        ])
        self.readout = nn.Linear(dims[-1], output_dim, bias=False)

    def init_latents(self, batch_size, device):
        return [
            torch.randn(batch_size, d, device=device,
                        requires_grad=False)
            for d in self.dims[1:]
        ]

    def compute_errors(self, inputs_latents):
        errors, gain_modulated_errors = [], []
        for l, layer in enumerate(self.layers):
            x_hat, a = layer(inputs_latents[l + 1])
            err = inputs_latents[l] - x_hat
            gm_err = err * layer.activation_deriv(a)
            errors.append(err)
            gain_modulated_errors.append(gm_err)
        return errors, gain_modulated_errors
```

The `__init__` constructor takes

$$\text{dims} = [d_0, d_1, \dots, d_L] \quad \text{and} \quad \text{output_dim} = d_{\text{out}}$$

specifying the dimensions of each layer in the network (from the input's d_0 to the topmost latent's d_L and readout's d_{out}) and performs the following:

- **Dimensions** (`self.dims`, `self.L`): The `dims` list and `self.L = len(dims)-1 = L` are stored.
- **Generative layers** (`self.layers`): A `torch.nn.ModuleList` is created to hold the stack of `PCNLayer` instances representing the L latent layers. Each `PCNLayer` maps $\mathbb{R}^{d_{l+1}} \rightarrow \mathbb{R}^{d_l}$ as in the generative model. The dimensions are drawn from `dims`.
- **Readout layer** (`self.readout`): For supervised learning, a `torch.nn.Linear` layer is defined as `self.readout`. As with the latent layers, the learnable weight matrix `self.readout.weight` corresponding to \mathbf{W}^{out} is initialized as Xavier uniform. This layer

maps the topmost latent state $\mathbf{X}^{(L)}$ (with dimension `dims[-1] = d_L`) to the output prediction $\hat{\mathbf{Y}}$ (with dimension `output_dim = d_{out}`). In alignment with the supervised extension described, the layer implements $\hat{\mathbf{Y}} = \mathbf{X}^{(L)} \mathbf{W}^{\text{out}\top}$ without a bias term.

The class includes two methods to facilitate the PCN’s operation:

- `init_latents(self, batch_size, device)`: This method initializes the latent variables as in lines 1–3 in Algorithm 3. For a given `batch_size`, it creates a list of tensors $[\mathbf{X}^{(1)}, \dots, \mathbf{X}^{(L)}]$ with values drawn independently from $\mathcal{N}(0, 1)$, ensuring they are on the specified `device`. The flag `requires_grad=False` ensures PyTorch’s autograd engine does not compute or store gradients for these latent variables, as they are updated via the explicit inference rule instead of automatic differentiation of a loss function.
- `compute_errors(self, inputs_latents)`: This method calculates the prediction errors $\mathbf{E}^{(l)}$ and the gain-modulated errors $\mathbf{H}^{(l)}$ for all layers $0 \leq l < L$. It takes a list `inputs_latents = $[\mathbf{X}^{(0)}, \mathbf{X}^{(1)}, \dots, \mathbf{X}^{(L)}]$` (containing the current input batch and all current latent states) as input. It iterates through each `PCNLayer` in `self.layers`, using the layer’s `forward` method to get the prediction $\hat{\mathbf{X}}^{(l)}$ and pre-activation $\mathbf{A}^{(l)}$. It then computes $\mathbf{E}^{(l)} = \mathbf{X}^{(l)} - \hat{\mathbf{X}}^{(l)}$ and $\mathbf{H}^{(l)} = \mathbf{E}^{(l)} \odot f^{(l)'}(\mathbf{A}^{(l)})$. These computations are central to both the inference and learning phases, as seen in lines 6–9 and 21–24 of Algorithm 3. The method returns two lists: one containing all $\mathbf{E}^{(l)}$ and another containing all $\mathbf{H}^{(l)}$.

Together, these components and methods provide a modular PyTorch representation of the predictive coding network, suitable for the supervised learning task described.

5.4.3 Training loop

Finally, to show the operational aspects of the model, a minimal training loop is presented next. Focusing on essential update mechanisms, it differs slightly from the accompanying Python notebook, which also implements the energy-tracking features discussed earlier.

The function `train_pcn` takes the PCN `model`, a PyTorch `data_loader` for batching, the number of training `num_epochs`, rates `eta_infer` and `eta_learn`, and the number of steps for inference (`T_infer`) per sample and learning (`T_learn`) per batch, along with the target `device`.

The training process begins by setting the `model` to training mode and moving it to the specified `device`. It then iterates for a given number of `num_epochs`. Within each epoch, mini-batches of data (`x_batch`, `y_batch`) are processed. Input features `x_batch` are flattened and, along with the one-hot encoded targets `y_batch`, moved to the target `device`. The list `inputs_latents` is initialized to hold the input batch $\mathbf{X}^{(0)}$ followed by the randomly initialized latent variables $\mathbf{X}^{(1)}, \dots, \mathbf{X}^{(L)}$ obtained from `model.init_latents()`. Notably, a list named `weights` is created containing direct references to the model’s weight tensors (`layer.W` for generative layers and `model.readout.weight` for the readout layer). Updates to elements of this `weights` list will therefore modify the model’s parameters in place.

Core training loop

```
def train_pcn(model, data_loader, num_epochs, eta_infer, eta_learn,
              T_infer, T_learn, device='cuda'):
    model.to(device).train()

    for epoch in range(num_epochs):
        for x_batch, y_batch in data_loader:
            B = x_batch.size(0)
            d_0 = model.dims[0]
            x_batch = x_batch.view(B, d_0).to(device)
            y_batch = F.one_hot(y_batch, num_classes=model.readout.out_features) \
                .float().to(device)

            inputs_latents = [x_batch] + model.init_latents(B, device)
            weights = [layer.W for layer in model.layers] + [model.readout.weight]

            # INFERENCE - T_infer steps
            with torch.no_grad(), autocast(device_type='cuda'):
                for t in range(1, T_infer + 1):
                    errors, gain_modulated_errors = model.compute_errors(inputs_latents)
                    y_hat = model.readout(inputs_latents[-1])
                    eps_sup = y_hat - y_batch
                    eps_L = eps_sup @ weights[-1]
                    errors_extended = errors + [eps_L]

                    # Latent gradients and updates
                    for l in range(1, model.L + 1):
                        grad_Xl = errors_extended[l] - \
                            gain_modulated_errors[l-1] @ weights[l-1]
                        inputs_latents[l] -= eta_infer * grad_Xl

            # LEARNING - T_learn steps
            with torch.no_grad():
                for t in range(T_infer + 1, T_learn + T_infer + 1):
                    errors, gain_modulated_errors = model.compute_errors(inputs_latents)
                    y_hat = model.readout(inputs_latents[-1])
                    eps_sup = y_hat - y_batch

                    # Weight gradients and updates
                    for l in range(model.L):
                        grad_Wl = -(gain_modulated_errors[l].T @ inputs_latents[l+1]) / B
                        weights[l] -= eta_learn * grad_Wl
                    grad_Wout = eps_sup.T @ inputs_latents[-1] / B
                    weights[-1] -= eta_learn * grad_Wout
```

For each batch, the **inference phase** is executed for T_{infer} steps. This phase operates under `torch.no_grad()` context, as latent variable updates are performed manually according to the PCN rules, not via PyTorch’s autograd. The `autocast` context manager is also used here, enabling mixed-precision computations on compatible CUDA hardware. In each inference step:

1. The `model.compute_errors` method is called to calculate the current prediction errors $\mathbf{E}^{(l)}$ and gain-modulated errors $\mathbf{H}^{(l)}$ for the generative layers ($0 \leq l < L$).
2. The supervised prediction $\hat{\mathbf{Y}}$ and error \mathbf{E}^{sup} are computed.
3. The error signal for the top latent layer, $\mathbf{E}^{(L)} = \mathbf{E}^{\text{sup}} \mathbf{W}^{\text{out}}$, is calculated. This notation aligns with how $\boldsymbol{\varepsilon}^{(L)}$ was defined for the supervised algorithm to use the general latent update rule. The full list of errors $[\mathbf{E}^{(0)}, \dots, \mathbf{E}^{(L)}]$ is assembled in `errors_extended`.

- Each latent variable $\mathbf{X}^{(l)}$ (for $1 \leq l \leq L$) is updated using the formula $\mathbf{X}^{(l)} \leftarrow \mathbf{X}^{(l)} - \eta_{\text{infer}}(\mathbf{E}^{(l)} - \mathbf{H}^{(l-1)}\mathbf{W}^{(l-1)})$, which corresponds to line 15–16 of Algorithm 3.

Following inference, the **learning phase** adjusts the model weights for `T_learn` steps, also under `torch.no_grad()`. The latent variables `inputs_latents` remain fixed at their values from the end of the inference phase. In each of these `T_learn` steps:

- The generative errors $\mathbf{E}^{(l)}$, gain-modulated errors $\mathbf{H}^{(l)}$, and the supervised error \mathbf{E}^{sup} are recomputed. This is essential because the model weights (elements of the `weights` list) are updated within this loop, and thus the predictions and errors change accordingly.
- The generative weight matrices $\mathbf{W}^{(l)}$ (for $0 \leq l < L$) are updated. The gradient $\mathbf{G}_{\mathbf{W}}^{(l)} = -\frac{1}{B}\mathbf{H}^{(l)\top}\mathbf{X}^{(l+1)}$ is computed, and the weights are adjusted: $\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta_{\text{learn}}\mathbf{G}_{\mathbf{W}}^{(l)}$. This corresponds to lines 25–26 of Algorithm 3.
- Similarly, the readout weights \mathbf{W}^{out} are updated using their gradient $\mathbf{G}_{\mathbf{W}}^{\text{out}} = \frac{1}{B}\mathbf{E}^{\text{sup}\top}\mathbf{X}^{(L)}$, as per lines 30–31 of Algorithm 3.

5.5 Results and observations

The model was trained in Google Colab on an NVIDIA L4 GPU, running the accompanying Python notebook [20]. Training for just 4 epochs took 4 minutes. Yet the test performance scores in Table 1 are phenomenal. In fact, the top-1 accuracy (percentage correct) comfortably tops the leaderboard on [Papers With Code](#) at the time of writing (May 29, 2025)—the previous record of 99.5% having been set by the vision transformer ViT-H/14 in 2020.

Top-1 accuracy	99.92%
Top-3 accuracy	99.99%

Table 1: PCN test accuracies after 4 epochs of training

The batch-averaged energy trajectories recorded during training are depicted in Figure 3.

Disclosure. The experiment was virtually one-shot. The rates were initially set to $\eta_{\text{infer}} = 0.1$ and $\eta_{\text{learn}} = 0.001$ for the very first run. Monitoring the batch-averaged energy trajectories during training, some batches displayed unstable inference while learning was generally slow. Consequently, the inference rate was cut in half to $\eta_{\text{infer}} = 0.05$ and the learning rate was quintupled to $\eta_{\text{learn}} = 0.005$ for the second run, which stabilized training and led to the test performance above on the first try. *Neither the model architecture nor the hyperparameters were tuned for performance in any way or at any point.* In order to prevent data-leakage from the test set to validation, the experiment terminated here. (Observe that CIFAR-10 does not include a separate validation set.) The trained weights are publicly available [20].

This little experiment demonstrates once more that predictive coding networks can be trained end-to-end on a basic vision task using local, biologically plausible update rules. While generally not yet competitive with state-of-the-art deep learning methods in terms of scope and, depending on task, raw accuracy, PCNs offer an intriguing model of computation with sound theoretical and practical motivations.

In view of the disclosure above, the author has no idea whether a much smaller architecture (or different hyperparameters) might achieve similar performance. We leave it to the curious reader to tune the model and its training for that perfect 100% score. Happy hunting! ☺

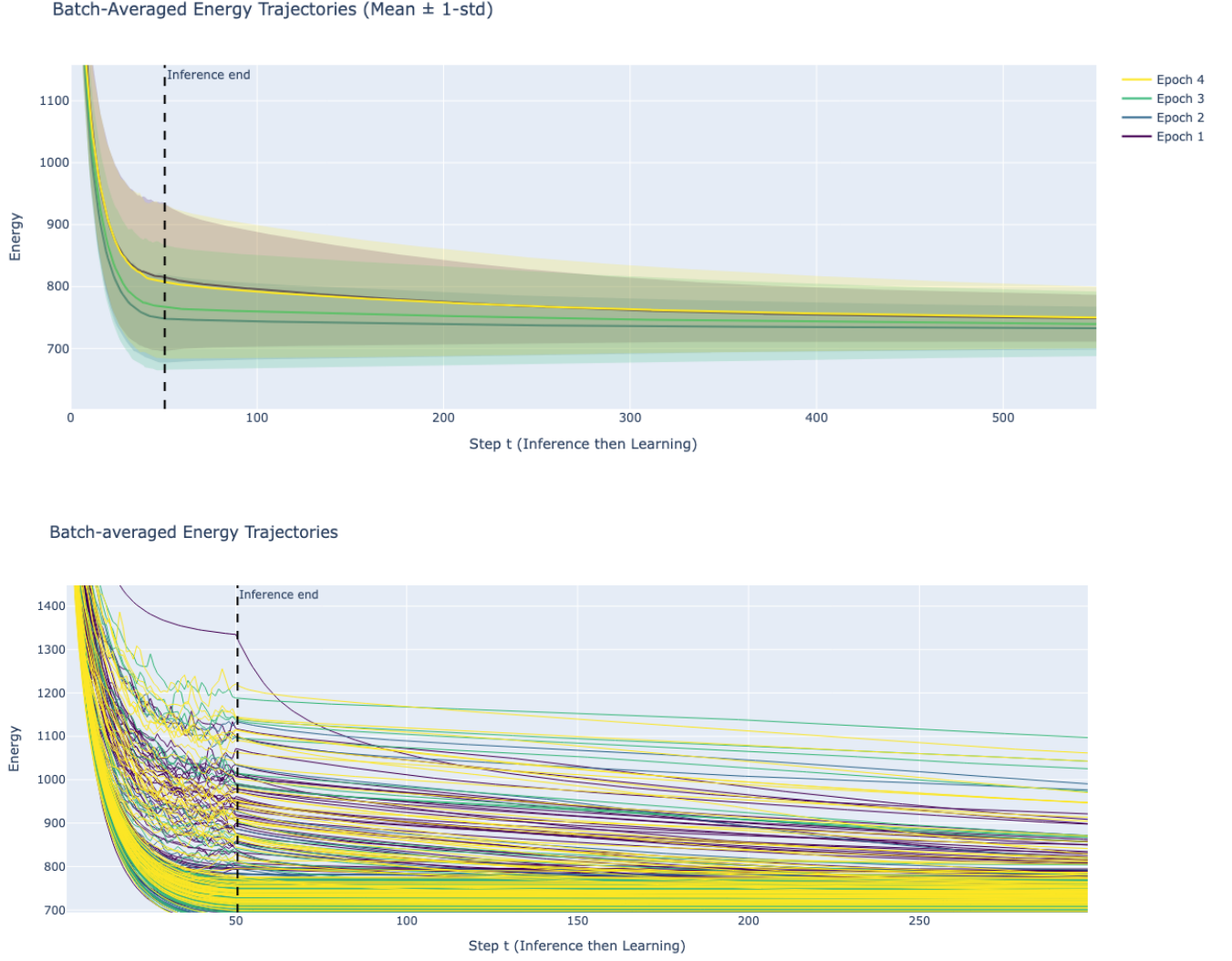


Figure 3: Partial zoom-in views of the batch-averaged energy trajectories recorded during training. The first plot shows the mean trajectories over all batches (equivalently, all samples) for each epoch, as well as the shaded areas of one standard deviation. The second plot shows the trajectories for individual batches. Coloring is by epoch. The full, interactive plots can be found in the Python notebook.

References

- [1] Nick Alonso. Predictive coding: A brief introduction and review for machine learning researchers, 2022. Blog post. URL: <https://neuralnetnick.com/2022/12/28/>.
- [2] Nick Alonso, Beren Millidge, Jeff Krichmar, and Emre Neftci. A theoretical framework for inference learning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, volume 36, pages 37335–37348, 2022. URL: <https://dl.acm.org/doi/10.5555/3600270.3602976>.
- [3] Horace B. Barlow. Possible principles underlying the transformation of sensory messages. In W. A. Rosenblith, editor, *Sensory Communication*, pages 217–234. MIT Press, Cambridge, MA, 1961.
- [4] Andre M. Bastos, W. Martin Usrey, Rick A. Adams, George R. Mangun, Pascal Fries, and Karl J. Friston. Canonical microcircuits for predictive coding. *Neuron*, 76(4):695–711, 2012. doi:10.1016/j.neuron.2012.10.038.
- [5] C. Caucheteux, A. Gramfort, and JR King. Evidence of a predictive coding hierarchy in the human brain listening to speech. *Nature Human Behaviour*, 7:430–441, 2023. doi:10.1038/s41562-022-01516-2.
- [6] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham China, Yongqiang Cao, Sri Harsha Choday, George Dimou, Harish Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018. doi:10.1109/MM.2018.112130359.
- [7] Simon Frieder and Thomas Lukasiewicz. (Non-)Convergence results for predictive coding networks. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 6793–6810, 2022. URL: <https://proceedings.mlr.press/v162/frieder22a.html>.
- [8] Karl Friston. A theory of cortical responses. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 360(1456):815–836, 2005. doi:10.1098/rstb.2005.1622.
- [9] Karl Friston. The free-energy principle: a unified brain theory? *Nature Reviews Neuroscience*, 11(2):127–138, 2010. doi:10.1038/nrn2787.
- [10] Richard L. Gregory. Perceptions as hypotheses. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 290(1038):181–197, 1980. doi:10.1098/rstb.1980.0090.
- [11] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018. URL: <http://arxiv.org/abs/1803.10122>.
- [12] Hermann von Helmholtz. *Treatise on Physiological Optics, Volume III: Concerning the perceptions in general*. Dover, New York, 1867. Translated by J. P. C. Southall, 3rd ed., 1962.
- [13] Georg B. Keller and Thomas D. Mrsic-Flogel. Predictive processing: A canonical cortical computation. *Neuron*, 100(2):424–435, 2018. doi:10.1016/j.neuron.2018.10.003.

- [14] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL: <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>.
- [15] William Lotter, Gabriel Kreiman, and David Cox. Deep predictive coding networks for video prediction and unsupervised learning. In *International Conference on Learning Representations (ICLR)*, 2017. URL: <https://arxiv.org/abs/1605.08104>.
- [16] Ankur Mali, Tommaso Salvatori, and Alexander Ororbia. Tight stability, convergence, and robustness bounds for predictive coding networks. 2024. doi:10.48550/arXiv.2410.04708.
- [17] Beren Millidge. Predictive coding as backprop and natural gradients, 2020. Blog post. URL: <https://www.beren.io/2020-09-12-Predictive-Coding-As-Backprop-And-Natural-Gradients/>.
- [18] Beren Millidge, Anil K. Seth, and Christopher L. Buckley. Predictive coding: a theoretical and experimental review. 2021. URL: <https://arxiv.org/abs/2107.12979>.
- [19] Beren Millidge, Alexander Tschantz, and Christopher L. Buckley. Predictive coding approximates backprop along arbitrary computation graphs. *Neural Computation*, 34:1329–1368, 2022. doi:10.1162/neco_a_01497.
- [20] Monadillo. An introduction to predictive coding networks for machine learning, 2025. GitHub repository containing the following supplements to this document: Python notebook and model weights. URL: <https://github.com/Monadillo/pcn-intro>.
- [21] David Mumford. On the computational architecture of the neocortex. ii. the role of cortico-cortical loops. *Biological Cybernetics*, 66(3):241–251, 1992. doi:10.1007/BF00198477.
- [22] Antony W. N’dri, William Gebhardt, Céline Teulière, Fleur Zeldenrust, Rajesh P. N. Rao, Jochen Triesch, and Alexander Ororbia. Predictive coding with spiking neural networks: a survey. 2024. URL: <https://arxiv.org/abs/2409.05386>.
- [23] Rajesh P. N. Rao and Dana 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. doi:10.1038/4580.
- [24] Tommaso Salvatori, Yuhang Song, Yordan Yordanov, Beren Millidge, Zhenghua Xu, Lei Sha, Cornelius Emde, Rafal Bogacz, and Thomas Lukasiewicz. A stable, fast, and fully automatic learning algorithm for predictive coding networks. In *International Conference on Learning Representations*, 2024. doi:10.48550/arXiv.2212.00720.
- [25] Jürgen Schmidhuber. Learning factorial codes by predictability minimization. *Neural Computation*, 4(6):863–879, 1992. doi:10.1162/neco.1992.4.6.863.
- [26] Stewart Shipp. Neural elements for predictive coding. *Frontiers in Psychology*, 7:1792, 2016. doi:10.3389/fpsyg.2016.01792.
- [27] Yuhang Song, Thomas Lukasiewicz, Zhenghua Xu, and Rafal Bogacz. Can the brain do backpropagation? — exact implementation of backpropagation in predictive coding networks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 22566–22579. Curran Associates, Inc., 2020. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/fec87a37cdeec1c6ecf8181c0aa2d3bf-Paper.pdf.

- [28] M.W. Spratling. Reconciling predictive coding and biased competition models of cortical function. *Frontiers in Computational Neuroscience*, 2:4, 2008. doi:[10.3389/neuro.10.004.2008](https://doi.org/10.3389/neuro.10.004.2008).
- [29] Alexander Tschantz, Beren Millidge, Anil K. Seth, and Christopher L. Buckley. Hybrid predictive coding: Inferring, fast and slow. *PLOS Computational Biology*, 19(8):1–31, 08 2023. doi:[10.1371/journal.pcbi.1011280](https://doi.org/10.1371/journal.pcbi.1011280).
- [30] K. S. Walsh, D. P. McGovern, A. Clark, and R. G. O’Connell. Evaluating the neurophysiological evidence for predictive processing as a model of perception. *Ann N Y Acad Sci*, 464(1):242–268, 3 2020. doi:[10.1111/nyas.14321](https://doi.org/10.1111/nyas.14321).
- [31] James C. R. Whittington and Rafal Bogacz. An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity. *Neural Computation*, 29:1229–1262, 2017. doi:[10.1162/NECO_a_00949](https://doi.org/10.1162/NECO_a_00949).