Performance Analysis of Hybrid Predictive Coding Compared to Classical Neural Networks

Jana Poľašková

January 4, 2026

Abstract

Hybrid Predictive Coding (HPC) has been proposed as a biologically inspired alternative to classical backpropagation, combining inference and learning into a single local process. In this project, we experimentally evaluate HPC and compare to fully connected neural networks trained on the MNIST dataset. The primary focus is on performance, training dynamics, and robustness-related properties, while theoretical background and implementation details are provided in the appendix. The results serve as a foundation for future work extending HPC to convolutional and transformer-based architectures.

1 Introduction

Deep neural networks are typically trained using backpropagation, a global learning algorithm that requires the propagation of error signals through the entire network. Despite its success, backpropagation has been criticized for its limited biological plausibility and robustness issues.

Hybrid Predictive Coding (HPC) offers an alternative learning in which inference and weight updates are tightly coupled and performed locally between adjacent layers. This project tries to explore whether these properties lead to differences in learning behavior and model performance when compared to a standard fully connected neural network trained with backpropagation. The main contribution of this work is an experimental comparison of HPC and a classical training.

2 Experimental Setup

The experiments are conducted using fully connected neural network and Hybrid predictive coding trained on the MNIST dataset. Both the HPC model and the classical baseline share **the same architecture** to ensure a fair comparison. HPC model training is taken from github library ((Tschantz et al., 2023)). Detailed explanation of Hybrid predictive coding with example of hybrid training is in a Appendix A.

2.1 MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset (LeCun et al., 1998) consists of 60,000 training and 10,000 testing grayscale images of handwritten

digits of size 28×28 pixels and 10 classes, corresponding to the digits from 0 to 9. Figure 1 shows samples from this dataset.

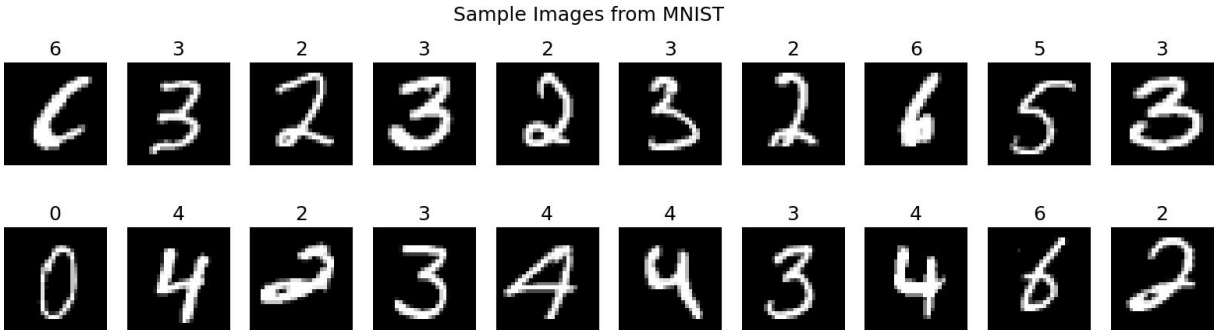


Figure 1: Sample images from the MNIST dataset

3 Fully Connected Neural network

The Fully Connected Neural Networks (FCNs) or Multilayer Perceptrons (MLPs) are a simple type of neural network where each neuron in a layer is connected to all the neurons in the next layer. They process input data as a single vector, making them effective for classification tasks.

An FCN consists of multiple layers with activation functions and an output layer that determines the target class. For training, it commonly uses backpropagation and optimization algorithms such as Adam or SGD to minimize a loss function.

3.1 Fully Connected Network for MNIST

This network consists of an input layer with 784 neurons (corresponding to the 28×28 pixel values of the images), followed by two hidden layers, each containing 500 neurons with Tanh activation functions. The output layer consists of 10 neurons, one for each digit class. For optimization, we used the Adam optimizer.

3.2 HPC Setup

Although the authors released the code and reported several key hyperparameters in the paper, reproducing the results turned out to be much more difficult than it initially appeared. The model is highly sensitive to its parameter settings, and small changes often led to very different training dynamics. Also the original codes only save the parameters and the final scores but dont save model and the weight itself, so this all needs to be modified.

To obtain stable results, I programmed hyperparameter search cycle, evaluating nearly all reasonable combinations of the main parameters (approximately 500 configurations in total, which takes almost two full days). Most configurations failed to converge and achieved very low final accuracy, while only a small subset reached accuracies above 60%.

Because this model differs fundamentally from standard neural networks, tuning it required a different intuition than usual, which made the process particularly challenging. In many cases, training became numerically unstable already after the first training

iteration. Specifically, when the learning rate was more than 0.003, several quantities frequently diverged and became NaN, including the amortised and predictive coding losses and errors.

I finally selected the parameter configuration reported in Table 1, which provided stable training and reproducible performance consistent with the trends reported in the original paper.

Table 1: Hybrid Predictive Coding configuration for MNIST experiments.

Parameter	**Value**
Dataset	MNIST (28×28)
Training samples	Full dataset
Number of epochs	36
Total training batches	3000
Batches per epoch	110
Batch size (train / test)	64 / 64
Architecture (PC)	$[10, 500, 500, 784]$
Architecture (amortised)	$[784, 500, 500, 10]$
Activation function	tanh
Use bias	Yes
Optimizer	Adam
Learning rate (θ)	1×10^{-4}
Learning rate (ϕ)	1×10^{-4}
Gradient clipping	30
Weight decay	1×10^{-4}
Weight normalization	Enabled
Inference step size (μ_{dt})	0.01
Inference iterations (train / test)	300 / 300
Inference threshold (train / test)	0.003 / 0.003
Initial state noise (σ)	0.005
Label scaling	0.94
Input normalization	Enabled

3.3 FCN Setup

For training the classical neural network, I decided to use the same model architecture that I previously employed in my bachelor's thesis on the MNIST dataset. This architecture is implemented using the `pytorch_lightning` library. I chose this framework mainly because, for larger networks, it provides faster training and handles many technical details automatically, such as parameter scheduling and training loops (also the code structure is more readable to me).

The optimizer and learning rate schedule are handled internally by the framework. The main training parameters and final results are reported in Table 2.

4 Results

This section presents the experimental results comparing.

Table 2: Fully connected neural network results on MNIST.

Parameter	**Value**
Dataset	MNIST (28×28)
Training samples	Full dataset
Number of epochs	25
Batch size	Default (PyTorch Lightning)
Architecture	$[784, 500, 500, 10]$
Activation function	tanh
Optimizer	Adam
Learning rate	Default (Adam), schedule step=10, $\gamma = 0.1$

4.1 Classification Performance

Table 3 shows the comparison between the classical fully connected network and the Hybrid Predictive Coding model on clean MNIST data.

Metric	FCN	HPC	Better / Winner
Test Accuracy (%)	96.53	85.7	FCN
Final Loss	0.11	0.47	FCN
Mean Max Probability (correct)	0.97	0.89	FCN
Mean Max Probability (incorrect)	0.70	0.59	HPC
Predictive Entropy	0.11	0.47	FCN

Table 3: Comparison of classification and uncertainty-related metrics between FCN and HPC on clean MNIST.

4.2 Test Accuracy

We can see that the final test accuracy for HPC model is lower than the accuracy usually reported for classical neural networks on the MNIST dataset. This result is expected because the model used in this work is a generative model. Its main goal is to learn and represent the data, not to directly maximise classification accuracy. For this reason, the final accuracy is lower than in purely discriminative models (Tschantz et al., 2023).

Because the HPC model does not optimise accuracy in the same way as a classical neural network, a direct comparison of these two models is difficult (something like comapring apples to oranges). In this sense, the models have different goals, and the comparison is not fully fair. However, since Hybrid Predictive Coding is often discussed as a possible replacement for standard backpropagation, this comparison is still necessary.

If the goal is only to achieve the best possible final prediction on test data, then a classical fully connected neural network is the better choice.

4.3 Final Loss

Test accuracy shows the average performance on the test dataset. The final loss mainly describes how the training converges and does not always reflect how confident the model is in its predictions. This metric evaluates how well the predicted probabilities match the true class.

A problem appears (again) because the HPC model does not use a standard softmax layer in the last layer. Instead, it uses a label scaling parameter (label_scale = 0.94) to represent class information. Because of this, the outputs cannot be directly interpreted as probabilities, and metrics cannot be computed in a straightforward way.

To solve this issue, temperature scaling was applied. Temperature scaling modifies the model outputs to control how confident the predictions are. A higher temperature produces softer probability distributions, while a lower temperature produces sharper and more confident predictions. We then used thos scaling to both models to ensure fair comparison.

As shown in Table 3, the FCN achieves a much lower final loss than the HPC model. This indicates that the FCN is better optimised for the classification objective, which is consistent with the main goal of model.

4.4 Mean Probability

We also use the mean maximum predicted probability to measure how confident the model is on average. This metric is especially useful when we separate correctly and incorrectly classified samples. In this way, we can see whether the model is too confident when it makes mistakes.

For correctly classified samples, the FCN reaches a mean maximum probability of 0.9894, while the HPC model reaches 0.8963. This shows that the FCN is more confident when it makes correct predictions.

For incorrectly classified samples, the FCN has a mean maximum probability of 0.7006, while the HPC model shows a lower value of 0.5931. So the HPC model is less confident when it makes mistakes.

4.5 Predictive Entropy

Predictive uncertainty is measured using entropy,

$$H(p) = -\sum_{c} p(c) \log p(c),$$

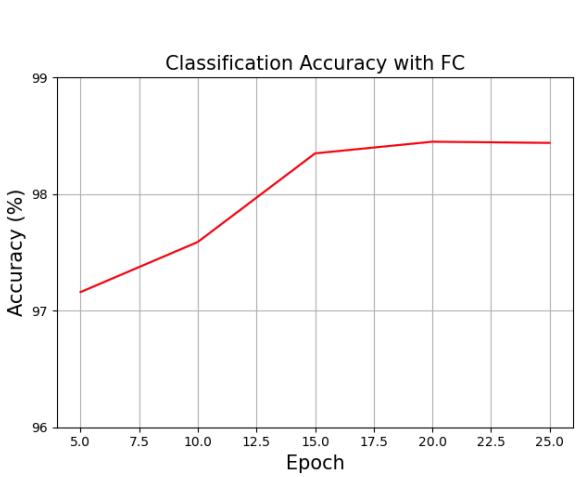
where $p(c)$ is the predicted probability of class c. Lower entropy means higher confidence and is usually observed for correct predictions.

The FCN achieves a low predictive entropy, reflecting sharp and confident predictions. In contrast, the HPC model exhibits a higher entropy, indicating greater uncertainty in its predictions. This behaviour is consistent with the generative nature of the HPC model.

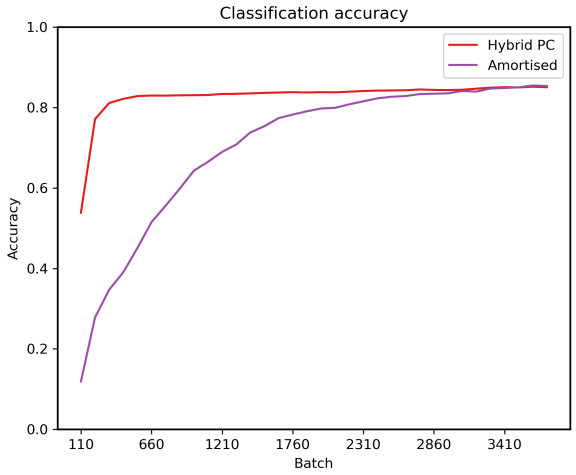
4.6 Training Behavior

In Figure 2a, we observe the training of the classical fully connected model. The model learns in a stable way, and we do not see any strong oscillations, divergence, or overfitting during training.

Figure 2b shows the training process of the HPC model. The hybrid model reaches its final performance level relatively quickly and then stays close to this level with only small improvements. The authors of the original paper also visualised the behaviour of the amortised model using the same weights and settings (I borrow these function so we share similar visualization). We can see that the amortised model needs more time to reach the desired performance level.



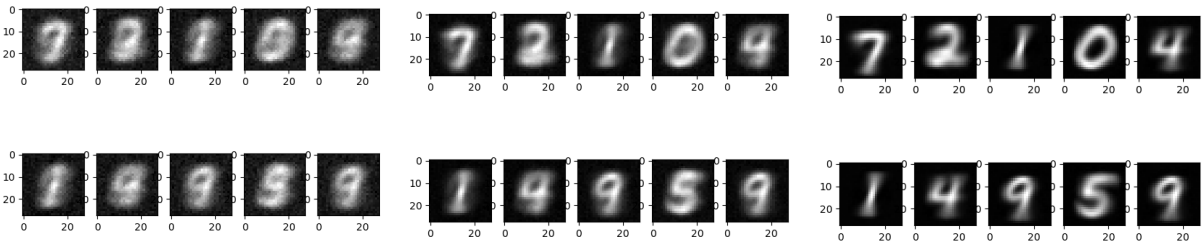
(a) Classification accuracy on the MNIST dataset for a fully connected neural network.



(b) Classification accuracy on the MNIST dataset for the Hybrid Predictive Coding architecture.

Hybrid predictive coding model use an iterative inference procedure. During this process, the amortised part of the model is trained to minimise the prediction error between its own predictions and the beliefs obtained after full iterative inference. For this reason, the accuracy of the amortised model cannot be higher than the accuracy achieved by iterative inference itself.

We observe stable training for both models, and neither model shows strong fluctuations. The training process is very similar. In the first epochs, the models quickly reach a near-final level of prediction, and later they mainly keep this performance and slightly improve it, without any visible decrease.



(a) Epoch 1 (b) Epoch 4 (c) Epoch 36

Figure 3: Illustrative samples taken from HPC throughout learning.

In Fig 3 we can see ilustrative samples from HPC learning. These images are generated by activating a single nodes in the highest layer (corresponding to a single digit), and performing top-down predictions in a layer-wise fashion. The images correspond to the predicted nodes at the lowest layer.

4.7 Training time

Training time is another useful parameter, since we want to optimise both performance and time. Since we used a fully connected model implemented with the well-optimised PyTorch Lightning library, this puts the FCN model at an advantage, when this framework has been optimised for many years.

In practice, this advantage was clearly visible. Training the FCN model to the reported performance took only 36 seconds. In contrast, training the HPC model required 1144 seconds, which is significantly longer.

4.8 Robustness Experiments

Since the HPC model did not perform very well in the standard classification test, I next tried a setting where it is often claimed to perform better. I added noise to the input images. In the first step, I used simple Gaussian noise, which means adding random noise to the image while keeping the noise strength bounded by the parameter σ.

Because both training and testing are performed on normalized images, the image must first be denormalized. After adding the noise, the image is normalized again using the same normalization parameters as before.

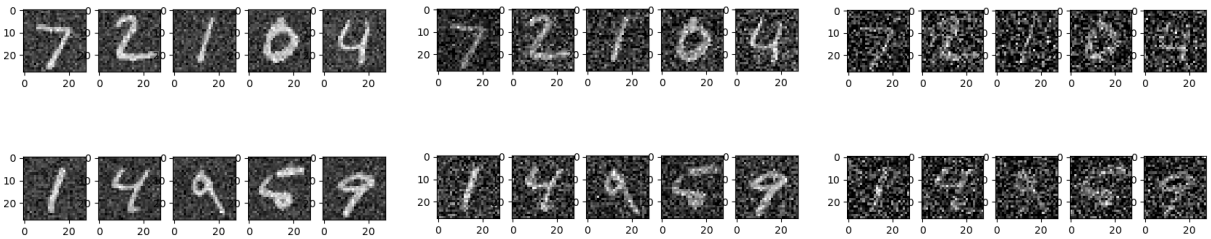
The results of this experiment are shown in Table 4.($\sigma = 0$ is being used just to comfired that results are the same as in the previous experiments)

However, the classification accuracy is still higher for the FCN model at all noise levels. Similarly, the predictive entropy is lower for the FCN model than for the HPC model, which indicates that the FCN produces more confident predictions. But this higher confidence is not always desirable, especially when the model makes incorrect predictions.

All other parameters, which are not shown here but are implemented in the code for this experiment, follow the same behaviour as in the original setting. They either improve or degrade according to the same trends observed for each model, and no additional conclusions beyond the original ones can be drawn from them.

Noise σ	Test Accuracy (%)		Predictive Entropy	
	FCN	HPC	FCN	HPC
0.00	96.54	85.68	0.11	0.47
0.05	95.47	80.63	0.14	0.63
0.10	90.83	70.63	0.29	0.93
0.20	68.09	48.40	0.96	1.54

Table 4: Robustness comparison under Gaussian noise for FCN and Hybrid HPC on MNIST.



(a) $\sigma = 0.05$ (b) $\sigma = 0.1$ (c) $\sigma = 0.2$

Figure 4: Examples of testing images under Gaussian noise

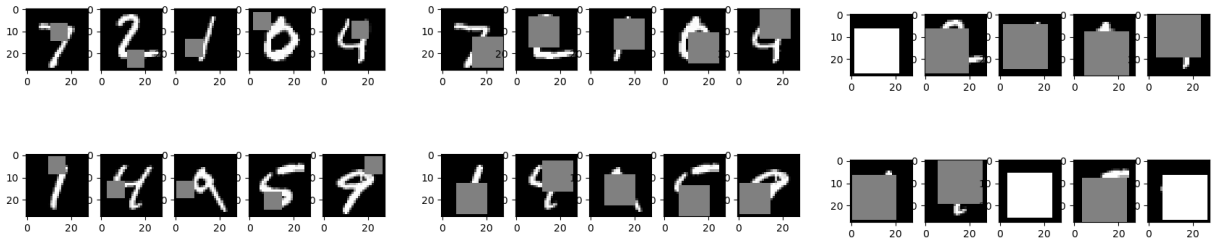
As a second experiment, I decided to use a standard occlusion approach, where a square of a fixed size is placed on a random part of the image. For each image, the position of the square is chosen randomly (the top and left corner), while ensuring that

the whole square fits inside the image. All pixels inside this region are then replaced by a constant value `fill` (fixed to 0.5).

The results can be seen in Table 5. Here, the HPC model finally performs better. For all occlusion sizes, HPC achieves higher accuracy and also lower, or very similar, predictive entropy. How the image look like we can see on Fig. 5. The occlusion with size 20 is already very large, so it more random guess than prediction.

Squre size	Test Accuracy (%)		Predictive Entropy	
	FCN	HPC	FCN	HPC
8	79.29	80.26	0.62	0.65
14	45.81	54.13	1.63	1.39
20	15.78	16.89	2.28	2.25

Table 5: Robustness comparison under square occlusion for FCN and Hybrid HPC on MNIST.



(a) Squares 8x8 (b) Squares 16x16 (c) Squares 20x20

Figure 5: Examples of testing images after adding random squares

5 Submitted Files

The github reposiratory of this project can be found:

https://github.com/JPolasko/hpc-comparison .

The main part of model training (calling other training functions) is located in the ‘project.py‘ file. Also every experiments which is presented in this work is located there.

More detailed training of the standard fully connected (FC) network is implemented in the ‘/standard_networks‘ directory. The neural network architecture is defined in ‘networks.py‘, and the complete training process is handled in ‘train.py‘. The training use dataset loading from ‘pybrid.datasets.py‘ to ensure same conditions.

More detailed training of the hybrid model is implemented in the remaining parts of the project, with the most important components located in the ‘pybrid‘ directory. Specifically, datasets are loaded in ‘dataset.py‘. The main training process is started from ‘scripts.py‘, which uses ‘optim.py‘, where the Adam optimizer is defined. Helper functions are located in ‘utils.py‘, and all required layer connections are defined in ‘layers.py‘. The ‘config.py‘ file contains the final architecture and parameters for the given network.

6 Conclusion and Future Work

As we can see, the fully connected model is still better in most aspects. The only case where the hybrid model performed better was in the robustness test with square occlusion added to the images. It should also be noted that since the HPC model does not use a standard softmax layer in the output layer, all of these comparisons were performed using temperature scaling. This approach may not fully reflect the results of both models.

From my personal experience while experimenting with the model and tuning its parameters, I observed that the final score of all the parameters of the HPC model changed significantly depending on the configuration.

For example, since this model does not focus directly on classification accuracy, the accuracy alone is not sufficient to evaluate its performance. The accuracy may be relatively good while other parameters are not. Therefore, during model tuning, it is not enough to optimize only accuracy; all relevant metrics must be considered together, which is much more difficult than in standard neural networks.

Based on this, I believe that there exist optimal parameter settings for this model that could achieve performance comparable to, or even better than, the fully connected network. However, such settings are not yet well established, and there are currently no clear guidelines for configuring this model, unlike classical fully connected networks, for which many well-tested recommendations exist.

Future work will extend this framework to convolutional architectures and investigate adversarial robustness in greater depth. Since I will continue with this analysis in my master thesis, I would welcome any feedback on this project, as well as suggestions and tips for further improvement.

References

Buckley, C. L., Kim, C. S., McGregor, S., and Seth, A. K. (2017). The free energy principle for action and perception: A mathematical review. *Journal of Mathematical Psychology*, 81:55–79.

Carlini, N. and Wagner, D. (2018). Audio adversarial examples: targeted attacks on speech-to-text. In *IEEE Security and Privacy Workshops*, page 1–7.

Cremer, C., Li, X., and Duvenaud, D. (2018). Inference suboptimality in variational autoencoders. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1078–1086. PMLR.

Fox, C. W. and Roberts, S. J. (2012). A tutorial on variational bayesian inference. *Artificial Intelligence Review*, 38(2):85–95.

Geirhos, R. et al. (2020). Shortcut learning in deep neural networks. *Nature Machine Intelligence*.

Grosse, K., Papernot, N., Manoharan, P., Backes, M., and McDaniel, P. (2017). Adversarial examples for malware detection. In *Computer Security – ESORICS 2017*, pages 62–79. Springer.

Ilyas, A., Santurkar, S., Tsipras, D., Engstrom, L., Tran, B., and Madry, A. (2019). Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems (NeurIPS) 32*.

Knill, D. C. and Pouget, A. (2004). The bayesian brain: the role of uncertainty in neural coding and computation. *Trends in Neurosciences*, 27(12):712–719.

Kurakin, A., Goodfellow, I. J., and Bengio, S. (2017). Adversarial examples in the physical world. In *Workshop track – ICLR 2017*.

LeCun, Y., Cortes, C., and Burges, C. (1998). The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J., and Hinton, G. E. (2020). Backpropagation and the brain. *Nature Reviews Neuroscience*, 21(6):335–346.

Millidge, B., Seth, A. K., and Buckley, C. L. (2022). Predictive coding: a theoretical and experimental review. *arXiv preprint arXiv:2107.12979*.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2014). Intriguing properties of neural networks. In *International Conference on Learning Representations*.

Thorpe, S., Fize, D., and Marlot, C. (1996). Speed of processing in the human visual system. *Nature*, 381(6582):520–522.

Tschantz, A., Millidge, B., Seth, A. K., and Buckley, C. L. (2023). Hybrid predictive coding: Inferring, fast and slow. *PLoS Computational Biology*, 18(11):e1011280.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2017). Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations (ICLR)*.

A Predictive Coding and Hybrid Predictive Coding

A.1 Limitations of Standard Neural Networks

The main limitation in neural networks is that they differ significantly from the biological mechanisms of the human brain. Unlike classical feedforward neural networks, the human cortex performs continuous recurrent inference based on local prediction-error minimization rather than global gradient backpropagation (Lillicrap et al., 2020).

Standard neural networks of any kind remain highly vulnerable to *adversarial attacks*—small, carefully crafted perturbations added to the input that cause the model to produce incorrect predictions with high confidence, even when the unperturbed input is classified correctly (Szegedy et al., 2014). This phenomenon has been demonstrated across multiple domains, including speech recognition (Carlini and Wagner, 2018), image classification (Kurakin et al., 2017), and malware detection (Grosse et al., 2017). Prior work has shown that adversarial examples are not bugs, they are features (Ilyas et al., 2019), suggesting the need for alternative, more robust computational models.

Another issue is their susceptibility to overfitting: deep networks often memorize training data instead of generalizing, leading to poor performance on new data. This can be interpret as they do not understand patterns in a human-like manner (Zhang et al., 2017).

They also rely on so-called *shortcut features—simple*, surface-level cues rather than meaningful abstractions. For example, if an image shows a cow on a beach, the model may fail to recognize the cow because it does not expect this combination of features and instead classifies it as something else (Geirhos et al., 2020).

A.2 Predictive Coding Framework

The framework assumes a multi-layered architecture in which each layer l maintains an internal state (activity) μ_l and generates a prediction of the state in the layer below, while the lowest layer attempts to predict the sensory input x itself. At every layer, two processes operate: an *iterative inference* step that updates neural activities, and a *learning* step that updates synaptic weights. Inference happens at every forward pass (fast), whereas learning occurs after inference converges (slow).

A.2.1 Approximate Bayesian Inference Motivation

First, we need to ensure that the model maintains a set of activities μ_l, that represent how the input is encoded on layer l. For images, these activities can be imagined as feature vectors describing how the image look on the current layer (Millidge et al., 2022).

In a standard neural network, information is propagated primarily by updating the weights. In contrast, in predictive coding, we first update these activities separately, before adjusting the weights. We achieve this by performing **approximate Bayesian inference** by continuously updating activities based on information from both the layer above and the layer below.

A similar principle is observed in the human brain. According to Knill and Pouget (2004), the brain tries to identify objects based on uncertain, noisy sensory inputs. Even when multiple different objects could produce the same pattern of retinal stimulation, the visual system needs to reconstructs the most plausible hidden cause (Millidge et al., 2022).

Let z denote sensory observations and x the hidden cause of this data. So we are trying to calculate probability of how likely it is x if z has occurred. The model assumes a generative distribution:

$$p(z, x) = p(z \mid x)\, p(x) \tag{1}$$

from which the ideal Bayesian posterior would be

$$p(x \mid z) = \frac{p(z \mid x)p(x)}{p(z)}. \tag{2}$$

where $p(z \mid x)$ is the likelihood, which tells us how likely the observed data z is given a hidden cause x. $p(x)$ is the prior distribution, describing the prior probability of hidden causes, and $p(z) = \int p(z \mid x)p(x)\, dx$ is the evidence, describing the overall probability of observing the data z considering all possible hidden causes.

However, computing the evidence $p(z)$ is typically intractable, since it requires integrating over all possible hidden causes (Buckley et al., 2017). This makes exact Bayesian inference computationally unrealistic.

To address this, predictive coding adopts **variational inference**. This method is also a biologically plausible for neural computation (Fox and Roberts, 2012). Variational inference introduces a tractable approximate posterior $q(x \mid z; \lambda)$ with parameters λ. The goal is to choose λ such that $q(x \mid z; \lambda)$ is as close as possible to the true posterior $p(x \mid z)$.

The difference between these two distributions is measured using the Kullback–Leibler (KL) divergence as a optimization problem:

$$D_{\mathrm{KL}}[q_\lambda(z) \,\|\, p(z \mid x)] = \mathbb{E}_{q_\lambda(z)} \left[\ln q_\lambda(z) - \ln p(z \mid x)\right] \tag{3}$$

Since $p(x \mid z)$ depends on the intractable evidence $p(z)$, we cannot minimize this quantity directly. Instead, we define the **variational free energy** $\mathcal{F}$ as

$$\begin{aligned} \mathcal{F}(z, x) &= \mathbb{E}_{q_\lambda(z)}[\ln q_\lambda(z) - \ln p(z, x)] \\ &= \mathbb{E}_{q_\lambda(z)}[\ln q_\lambda(z) - \ln p(z \mid x)] - \ln p(x) \\ &\geq D_{\mathrm{KL}}[q_\lambda(z) \,\|\, p(z \mid x)]\,. \end{aligned} \tag{4}$$

Minimizing the variational free energy $\mathcal{F}$ ensures that $q_\lambda(z)$ approaches the true posterior, thus implementing an approximate form of Bayesian inference.

$$\lambda^* = \arg\min_\lambda \mathbb{E}_{q_\lambda(z)} \left[\ln q_\lambda(z) - \ln p_\theta(z, x)\right] \tag{5}$$

The parameters λ can then be updated by standard optimization methods, such as gradient descent.

Heuristically, for each new input z, iterative inference initializes the variational parameters λ and then updates them step by step to minimise $\mathcal{F}(z, \lambda)$.

A.2.2 Iterative Inference

In neural network implementations, we do not work with an explicit variational distribution $q(x \mid o; \lambda)$, instead the variational parameters are identified with the neural activities themselves. The activities μ_l at each layer play the role of the variational parameters λ, and inference proceeds by updating these activities through gradient descent.

The variational free energy reduces to a sum of local prediction errors ε:

$$\mathcal{F} = \sum_l \|\varepsilon_l\|^2, \quad \text{where} \quad \varepsilon_l = \mu_l - f_{l+1}(\mu_{l+1}) \tag{6}$$

This is the same free-energy objective as in variational Bayes, simply expressed in the structure of a neural network. Each layer l has an activity vector μ_l. The difference between the actual activity μ_l and the predicted activity $f_{l+1}(\mu_{l+1})$ defines the prediction error.

Note: In the full variational formulation (Millidge et al., 2022), each layer maintains not only a mean μ_l but also a covariance Σ_l, which introduces an additional term $\ln \det \Sigma_l$ in the free-energy expression. This term reflects the uncertainty of the approximate posterior. In neural predictive-coding models, the covariance is typically fixed to the identity, causing the log-determinant term to vanish.

Each layer then updates its activity locally using gradient descent on free energy:

$$\Delta\mu_l = \eta \left(\varepsilon_l - \varepsilon_{l-1} \frac{\partial f(\mu_l)}{\partial \mu_l}\right), \tag{7}$$

where η is a learning rate. The first term, ε_l, forces the activity to move toward the top–down prediction provided by the layer above. The second term, $\varepsilon_{l-1} \frac{\partial f(\mu_l)}{\partial \mu_l}$, pushes the activity to change in such a direction that it generates a better bottom–up prediction for the layer below.

Intuitively, we can think of the unit as trying to find a compromise between causing error by deviating from the prediction from the layer above, and adjusting their own prediction to resolve error at the layer below.

A.2.3 Amortised Inference as an Alternative to Iterative

Instead of directly updating the variational parameters λ via gradient descent, amortised inference introduces a parameterised function $f_\phi(x)$ that predicts these parameters in a single forward pass. This method is better suited for rapid processing

The parameters ϕ of the amortised inference model are optimised across the entire dataset, such that the predicted variational parameters $\lambda = f_\phi(x)$ approximate the solution that would otherwise be obtained through iterative inference. Once learned, inference no longer requires repeated optimisation steps and can be performed efficiently using a single feedforward computation. (Tschantz et al., 2023)

However, amortised inference cannot adapt inference to individual inputs and therefore suffers from the amortisation gap (Cremer et al., 2018), a loss in performance caused by sharing inference parameters across the dataset.

A.2.4 Weight learning

While iterative inference focuses on updating the variational parameters λ. The second step concerns the optimization of the model weights θ after the activities have converged to the optimum.

In the formulation of Tschantz et al. (2023), weight learning is performed by taking the gradient of free energy with respect to the generative weights. The update rule has the following form:

$$\Delta\theta_l = \eta \left(\varepsilon_l \frac{\partial f_l(\mu_l)}{\partial \theta_l} \right), \tag{8}$$

where we are again try to minimize the same free-energy function 6.

For a common neural network, where the generative mapping is linear $f(\mu_l) = W_l \mu_l$, the derivative simplifies and turns into a purely local Hebbian update.

$$\frac{\partial f(\mu_l)}{\partial W_l} = \mu_l^\top, \tag{9}$$

$$\Delta W_l = \kappa\, \varepsilon_l\, \mu_l^\top. \tag{10}$$

This learning rule is biologically possible because each synapse connecting neuron j in layer l to neuron i in layer $l-1$ is updated using only the local prediction error of the postsynaptic unit and the activity of the presynaptic unit, $\mu_{l,j}$.

The two processes jointly minimise the same free-energy objective, but operate on different timescales: activities update rapidly for each input, while weights update slowly across inputs.

In this hierarchy, bottom-up signals carry prediction errors—the mismatch between predicted and observed activity—while top-down signals transmit predictions about expected representations. Learning emerges by minimizing these errors locally at each layer.

Although variations of predictive coding exist, they share this core principle of combining top-down predictions with bottom-up error correction.

A.3 Hybrid Predictive Coding

Hybrid predictive coding was originally introduced by Tschantz et al. (2023) to address the time and memory complexity of standard predictive coding.

Compared to standard predictive coding, hybrid predictive coding extends the model by adding bottom-up predictive mappings that learn to generate latent beliefs at higher layers. These bottom-up predictions are trained to match the beliefs produced by iterative inference and therefore learn to approximate posterior representations. In this framework, bottom-up processing can be understood as amortised inference, while top-down processing continues to play the role of iterative predictive coding. As a consequence, both bottom-up and top-down signals carry information about neural activity as well as prediction errors, in contrast to standard predictive coding, where bottom-up signals mainly transmit prediction errors and top-down signals transmit predictions.

A.3.1 Motivation for Hybridizing

Standard predictive coding performs inference exclusively through iterative minimization of prediction errors. This method requires multiple sequential inference steps for each new input. The main idea of learning is that a successful perception requires multiple cycles of neural activity.

Here, we encountered the computational limitations as well as empirical evidence showing that several forms of visual perception—such as gist perception or basic object recognition—occur extremely rapidly, on timescales that do not allow for substantial recurrent activity (Thorpe et al., 1996).

Here comes the HPC to resolve this problem. It combines amortised (fast) inference with iterative (slow) predictive coding within a single architecture and under a single variational objective. Since amortised inference operates very quickly, while iterative inference requires repeated computation and therefore operates on a much slower timescale, these two approaches are often referred to as fast and slow inference, respectively.

Tschantz et al. (2023) claim that models have an increased degree of biological relevance also.

A.3.2 Amortised Initialization and Iterative Predictive Coding

HPC integrates amortised (section A.2.3) and iterative inference (section A.2.2) within a single predictive coding architecture. As in standard predictive coding, the model consists of a hierarchy of layers, each associated with latent activities μ_l and prediction errors ε_l. Top-down connections parameterised by θ generate predictions of lower-layer activity, while prediction errors quantify the mismatch between predictions and observed activity.

In addition to the generative top-down pathways, HPC introduces an amortised bottom-up network parameterised by ϕ. These bottom-up connections implement feedforward mappings that predict higher-layer activities from lower-layer activity, including a direct mapping from sensory input at the lowest layer. Both the top-down and bottom-up pathways therefore aim to predict the same latent variables μ_l, but operate in opposite directions.

Learning in the hybrid model follows the same free-energy minimisation principle as in standard predictive coding (section A.2.4), and is applied to both the generative (θ) and amortised (ϕ) parameters.

A.3.3 Hybrid Training

Inference proceeds in two stages. First, an amortised feedforward pass propagates sensory input up the hierarchy using the bottom-up mappings, producing an initial estimate of the latent activities. This stage provides a fast approximation to inference. Second, these initial activities are refined through iterative predictive coding, during which top-down predictions and local prediction errors are used to minimise variational free energy.

After inference converges, both sets of parameters are updated. The generative parameters θ are learned using the converged activities, as in standard predictive coding. The amortised parameters ϕ are trained to predict these converged activities by minimising the discrepancy between the amortised predictions and the final inferred activities. In this way, the amortised network gradually learns to approximate the outcome of iterative inference, enabling increasingly accurate and efficient inference through feedforward processing.

Algorithm 1 briefly demonstrates the structure of training model using hybrid predictive coding, as introduced by (Tschantz et al., 2023).

Algorithm 1 Hybrid Predictive Coding

Require: Generative parameters θ, amortised parameters ϕ, data x, step size κ, learning rate α

Amortised Inference:
$\mu_0 \leftarrow f_0^\phi(x)$
for $i = 0 \dots L-1$ **do**
    $\mu_{i+1} \leftarrow f_i^\phi(\mu_i)$
end for
Iterative Inference:
for $j = 1 \dots N$ **do** ▷ optimisation iterations
    $\varepsilon_l \leftarrow x - f_0^\theta(\mu_0)$
    $\varepsilon_p \leftarrow \mu_0 - f_1^\theta(\mu_1)$
    $\dot{\mu}_0 \leftarrow \kappa \left(\varepsilon_p \frac{\partial f^\theta(\mu_0)^\top}{\partial \mu_0} - \varepsilon_l \right)$
    for $i = 1 \dots L$ **do**
        $\varepsilon_l \leftarrow \mu_{i-1} - f_i^\theta(\mu_i)$
        $\varepsilon_p \leftarrow \mu_i - f_{i+1}^\theta(\mu_{i+1})$
        $\dot{\mu}_i \leftarrow \kappa \left(\varepsilon_p \frac{\partial f^\theta(\mu_i)^\top}{\partial \mu_i} - \varepsilon_l \right)$
    end for
end for
Learning of Parameters:
for $i = 0 \dots L$ **do**
    $\varepsilon_l^\phi \leftarrow \mu_{i+1}^* - f_i^\phi(\mu_i)$
    $\dot{\theta}_i \leftarrow \alpha \left(\varepsilon_l \frac{\partial f_i^\theta(\mu_i)^\top}{\partial \theta_i} \right)$
    $\dot{\phi}_i \leftarrow \alpha \left(\varepsilon_l^\phi \frac{\partial f_i^\phi(\mu_i)^\top}{\partial \phi_i} \right)$
end for