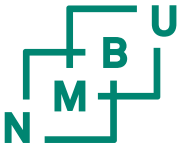


Norwegian
University of
Life Sciences

Master's Thesis 2024 30 ECTS
Faculty of Science and Technology

Feedback connections in neural networks for layer reuse

Hallvard Høyland Lavik
Data Science

Foreword

Thank you, Kristian Hovde Liland and Hans Ekkehard Plesser, for meeting with me every week. I appreciate the time and expertise you dedicate to these meetings.

This has been fun.

Just imagine: in the nerves, in the head, I mean there, in the brain, are these nerves (and little devils they are, too!)... you have these wiggly little tails, the nerves have little tail-endings, well, they only have to start thrashing about... in other words, look, I focus my eyes on something, like so, and they begin to thrash about, those little tails... and as soon as they start thrashing about, that's when the image appears, [...] *so there you have it—first I see, and only then do I think... it's on account of the little tail-endings, and not at all because I've a soul or that I'm some kind of image or likeness* [of God]*, that's all rubbish.*

Fyodor Dostoevsky in *The Brothers Karamazov*

Abstract

The increasing interest and integration of artificial intelligence (AI) have resulted in expanded artificial neural network (ANN) sizes following the proposed scaling laws [1–3] regarding their performance. Consequently, to accommodate practical deployment and widespread accessibility, the deep learning (DL) field and therein ANNs could benefit from revised architectures. Drawing inspiration from the biological brain and recurrent neural networks (RNNs), we propose feedback connections in ANNs, leveraging existing weights by looping subsequent outputs back to earlier layers. In this manner, networks can further iterate on their latent space representations of inputs without introducing additional trainable parameters. Here, we express novel approaches to implement said connections in ANNs with two different ways of backpropagation through these. Through the custom-built `neurons` framework [4], we easily evaluate and compare variations of feedback connections. While the framework has certain constraints regarding computational efficiency for large networks or datasets, we successfully demonstrate the benefits of feedback connections for specific problem settings. Furthermore, we probe these trained networks, thus assessing the performance impact and information flow of feedback loops. As we show promising results on small datasets, we expect this to scale accordingly, following the success of similar studies [5, 6]. Consequently, as networks implementing feedback connections mimic deeper networks while retaining a relatively small parameter footprint with comparable or even improved performance, we suspect upscaling our novel approaches to implement this may make the future of AI more accessible and deployable.

Sammendrag

Den økende interessen for og integreringen av kunstig intelligens (AI) har resultert i stadig større kunstige nevrale nettverk (ANN) som en direkte konsekvens av de antatte skaleringslovene [1–3]. For å tilrettelegge for praktisk implementering og økt tilgjengelighet, kan feltet, *i.e.*, dyplæring (DL), og derunder ANN ha fordel av reviderte arkitekturer. Ved å hente inspirasjon både fra den biologiske hjernen og gjentakende nevrale nettverk (RNN), introduserer vi tilbakekoplinger i ANN, hvilket utnytter eksisterende vekter ved å sende tidsmessig senere verdier tilbake til titdligere lag. På denne måten har nettverket mulighet til å videre iterere på sine latente representasjoner av inndataen, uten at trenbare parametre økes. Her presenterer vi nye tilnærminger for implementasjon av slike tilbakekoblingssløyfer, med to ulike måter å trene disse. Gjennom det egenutviklede `neurons`-rammeverket [4] kan vi enkelt evaluere og sammenligne ulike varianter av tilbakekoblinger. Mens rammeverket har visse begrensninger når det gjelder effektivitet for store nettverk eller datasett, lykkes vi med å demonstrere fordelene med tilbakekoblinger for bestemte problemstillinger. Videre kan vi undersøke disse trente nettverkene for å vurdere ytelsespåvirkningen og informasjonsflyten i disse sløyfene. Ettersom vi viser lovende resultater på små datasett, forventer vi at dette vil skalere tilsvarende, i tråd med suksessen fra lignende studier [5, 6]. Følgelig, siden nettverk som implementerer tilbakekoblinger etterligner dypere nettverk mens de beholder et relativt lite parameterfotavtrykk, med sammenlignbar eller til og med forbedret ytelse, mistenker vi at oppskalering av våre nye tilnærminger for implementering av disse kan føre til at fremtidens AI blir mer tilgjengelig og implementerbar.

Contents

Figures

Algorithms

Listings

Tables

Acronyms

Adam adaptive moment estimation.

AI artificial intelligence.

ANN artificial neural network.

DL deep learning.

EMSC extended multiplicative signal correction.

FFN feed-forward network.

FTIR fourier-transform infrared spectroscopy.

LLM large language model.

LSTM long short-term memory.

MSE mean squared error.

RMSE root mean squared error.

RMSprop root mean squared propagation.

RNN recurrent neural network.

SG Savitzky-Golay.

SGD stochastic gradient descent.

SOTA state-of-the-art.

1 Introduction

1.1 Motivation

The rapid growth of AI through ANNs has led to its widespread integration across various digital products, ranging from playing games to generating art and driving cars to advanced security systems [7–10], not to mention large language model (LLM) chatbots [11–13]. As a result, the development of ANNs has experienced proportional growth in the past decade.

The race to produce state-of-the-art (SOTA) AI products has resulted in increasingly large and complex ANNs, with research dominated by major technology corporations including Google, Meta, and Microsoft [14–18]. The continual growth in size and complexity of DL ANNs is largely driven by the scaling laws described in [1–3], suggesting a strong correlation between ANN size and performance. While the expansion of ANN architectures has enhanced their performance, it has simultaneously introduced significant challenges in cost-effectiveness and practical deployment.

As a result of the increasing scale of ANNs, methods like quantisation [19, 20] and low-rank adaptation [21], among others [22, 23], are being developed to combat the challenges of practical deployment. While model compression leads to lower latency, it typically[1] comes at the cost of reduced intelligence [24]. Therefore, to reduce the model sizes and retain (or even improve) intelligence, the field could benefit from a revised model architecture [25].

1.2 Objective

The biological brain is a highly complex network of intertwined connections [26]. By drawing inspiration thereof, the current aforementioned limitations of ANNs might be improved. While AI has its initial roots in neuroscience [27], the field (disregarding the more biologically plausible variants, here focusing on generally available SOTA solutions) has greatly deviated from its biological counterpart. Where ANNs (generally) processes information in a feed-forward manner, the brain consists of an intricate system with looping connections. Theories regarding consciousness point to this looping property as grounds for the conscious experience [28, 29].

[1] Depending on the specific method of quantisation [24].

With this in mind, this thesis aims to experiment with a somewhat more biologically plausible ANN structure to better leverage the available weights by introducing feedback loops resembling the complicated workings of a biological brain [26]. The thesis aims to study something in-between an feed-forward network (FFN) and an RNN. Mainly, the possibility of looping specific later layer outputs back to earlier layers of the network, possibly improving information extraction and reasoning by allowing the network to iterate its latent space representations. Moreover, we can imitate deeper networks with effectively fewer (trainable) parameters by adding feedback connections.

In the thesis, we propose novel ways of implementing feedback connections in ANNs with two different ways of backpropagating through these.

1.3 Related work

Recurrent neural networks

Distinguishable from our approach, while somewhat related, is the autoregressive RNNs [30–32]. Through RNNs, we acknowledge that weights are reused (coupled) across time lags. However, as these were developed to tackle issues related to time series, they differ from our networks, which are purely one-to-one concerning how data is handled.

However, during the forward pass, the unwrapping of our feedback connections compared to RNNs bears the exact general resemblance. Where RNNs use their hidden *memory* when processing subsequent inputs, our network instead directly reuses the latent representation (hidden state) and iterates purely on this. Of course, our network may combine the hidden state with the initial input, as we will describe more in detail, further increasing the resemblance between RNNs and feedback connections.

Concerning the backward pass, however, our approach differs from that of RNNs. Where RNNs incorporate backpropagation through time [33], we instead propose two novel approaches, described in Chapter 3.

Recurring residual blocks

Although two studies by Liao and Poggio [5] and Caswell, Shen, and Wang [6] have experimented with a somewhat similar structure to ours, their approach for backward propagation of errors is more comparable to that of RNNs. Here, they initialise the subsystems of the feedback connections with identical weights and accumulate all gradients before weight updates on the compacted representation, ensuring coupled weights at all times. Our forward pass is comparable to their implementation, but our backward implementations apply different techniques, further investigated in Chapter 5.

Recursive transformers

Recently, a study of reusable layers in LLMs was published [34]. Paving the way for future compression of pre-trained LLMs and methods for training these from scratch, the study presents similar methods as this thesis, albeit at a larger scale and purely concerning LLMs.

Entropy feedback

Additionally, worth mentioning is the concept of entropy feedback [35]. Here, the incorporation of strengthening certain connections of ANNs based on an entropy measure is studied. Specific node values are fed back, potentially enhancing the respective pathways.

2 Theory

A consequence of the increasing incorporation of AI into consumer products is the increased widening of DL research. In this chapter, we will define the underlying theory necessary for further study of the field in general terms. By first exploring the inspiration behind current AI, we understand how things came to be as they are today. Although we omit certain aspects of ANNs, we present its core driving elements, which we build upon in the subsequent chapter.

2.1 Biological neural networks

Before describing its artificial counterpart, it is worth mentioning the intricate workings of biological neural networks. While their differences are significant, the general inspiration behind present AI stems from the biological brain [27], and information processing in ANNs can, in simplified terms, be seen to mimic biological networks.

2.1.1 Neurons

Biological neurons, or nerve cells, are the fundamental units responsible for information processing in the brain. Although neurons vary in form and function (with new types still being found [36]), they generally share a standard structure, as illustrated in Figure 2.1. Each neuron has a soma (cell body) from which a single axon extends, responsible for transmitting information (signals) to the dendrites of other neurons [26]. Dendrites, branching out of the soma, establish connections between neurons through the synaptic cleft, where one neuron's axon meets another's dendrite.

Information flow between neurons occurs through ionic currents and molecular transfer [26]. We observe an ionic current through the neuron's axon during signal transmission. At the synaptic cleft, the pre-synaptic neuron (the neuron transmitting the signal) releases neurotransmitters that bind to receptors in the dendrites of the post-synaptic (receiving) neuron. Following synaptic input from many neurons, the receiving neuron can become *excited*, initiating its own signal.

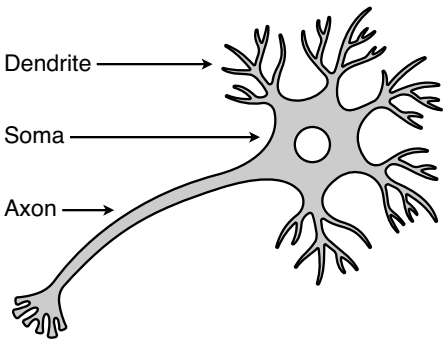


Figure 2.1: A simplified biological neuron.

Accumulation of inputs

Importantly, each neuron has a certain potential threshold concerning excitation [26]. Therefore, neuronal information propagation may only occur following sufficient excitable inputs. Different neurotransmitters have varying effects on the post-synaptic neuron, some promoting excitation while others inhibit it, thus affecting further signal transmission.

A neuron collects and accumulates ionic potential following incoming connections of many neurons. For any neuron to propagate a signal, this accumulated potential must reach some threshold value [26]. That is, given the influx of ionic current as a consequence of incoming signals, leading to a potential increase, this accumulated potential directly influences further signal propagation by the recipient.

Temporal processing

Additionally, the timing of signal transmission between two neurons is crucial for effective communication within the brain. In simplified terms, if neuron A receives a signal from neuron B moments *before* its subsequent signal propagation, it is assumed [26] that their relation is significant.

Oppositely, incoming signals to neuron A moments *after* its signal is sent are assumed to be insignificant. In this manner, the brain dynamically modifies strengths (*i.e.*, weights) of connections, enhancing significant ones and diminishing insignificant ones.

2.1.2 Networks

The brain, a network of interacting neurons, exhibits extensive dynamic coupling. Neurons propagate information through their axons to multiple recipients following their influx of information from a significantly greater number of neurons. In essence, a single neuron collects information from numerous neighbouring neurons, processes

this information, and potentially propagates a signal.

In turn, clusters of (or even single) neurons and their interactions with their environment (the brain) lead to changes in molecular densities [26], which affects signal propagation and, consequently, synapses. That being said, there is no one rule governing synapse strength modification, as this is a result of multiple factors.

While every brain is different from one another [37], research points towards a high occurrence of feedback connections [38, 39]. Interestingly, it is found that feed-forward connections outnumber feedback connections for short-distanced connections and that feedback connections are dominant for longer-distanced connections. In this manner, brain activity is constantly being held at a certain level, with signals propagating between neurons and clusters of neurons at all times.

It is, however, worth noting that the actual behaviour of the brain is far more complex, with, *e.g.*, neurotransmitter substance playing a crucial role, *et cetera*. Consequently, the functioning of a human brain, comprising approximately 86 billion neurons with this complex interconnectivity, is extraordinarily intricate and not yet fully understood [26].

2.2 Artificial neural networks

Although abstract similarities exist between artificial and biological neurons, SOTA ANNs differ significantly from their biological counterparts. The distinction between the two primarily arises from the need for ANNs to be both deployable and computationally efficient. While more biologically accurate ANNs exists [40–42], this thesis focuses on SOTA algorithms and networks that prioritise practical implementation over biological accuracy.

Figure 2.2 presents a simplified artificial neuron or node. While this representation is incomplete, as we will see in the subsequent section, it captures the fundamental structure that forms the building blocks of ANNs. The incoming and outgoing arrows represent connections with other nodes in a network. Each connection includes a strength, or *weight*, $w_i^{\{\text{in,out}\}}$, which directly modulates the propagated signal. This weight-based signal is analogous (although highly simplified) to the synaptic strengths between biological neurons.

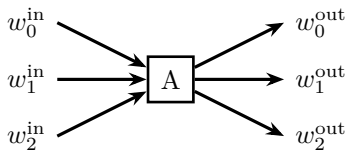


Figure 2.2: Equivalence of the biological neuron in ANNs.

Furthermore, each node (*i.e.*, A) holds a scalar value, and interactions between nodes can be implemented as a weighted sum of their connections. In this manner, we can denote the value of our node as

$$a_A = \sigma\left(\sum_{i=0}^{n}(w_{i,A}a_i) + b_A\right) \tag{2.1}$$

where each value of the preceding nodes, a_i, is multiplied with the corresponding weight connecting it to A, w_i^{in}. A bias term, b_A, is added, analogous to the threshold potential for neuronal activation in biological systems. The newly calculated value is thereafter processed through an *activation function*, $\sigma(\cdot)$, which we will detail further in the subsequent section.

By applying a deterministic set of rules on continuous values, ANNs leverage gradient-based optimisation techniques, essential for training complex SOTA ANNs effectively [41, 43]. In contrast, biological neurons operate through binary action potentials and, consequently, changes in ionic flows. While this behaviour is present in (the more biologically plausible) spiking neural networks [40], current SOTA ANNs generally use continuous values along with well-established learning methods, which we will come back to. This design choice of ANNs thus prioritises computational feasibility and practical deployment while abstractly resembling biological networks.

That being said, ANNs (as the name implies) have their roots in the study of their biological counterparts, drawing inspiration from the structure and functionality of the brain [27, 43]. While current SOTA ANNs are distinctly different from their organic counterparts, bridging the gap between the two might help further improve the current capabilities seen in ANNs [5, 35].

2.3 Feed-forward neural networks

While SOTA ANNs exhibits complex processing of data [44–47], all models can (at least at a smaller, abstracted scale) be seen to process information procedurally. With this in mind, we define the FFN, which represents these more minor parts of SOTA networks or entire networks of smaller scale. Thus, the terms ANN and FFN are used interchangeably, following their equivalent base case interpretation.

The described procedural information flow arises from how FFNs processes information. Namely, FFNs repeatedly feeds values forwards through its layers [43]; given some input values, the network projects these through its layers, yielding an output. While FFNs might differ significantly from one another concerning size and function, they all share the core structure seen in Figure 2.3.

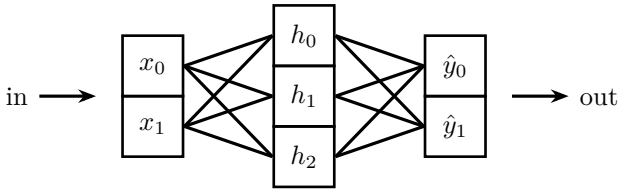


Figure 2.3: A simple ANN of three layers.

Through Figure 2.3, we present an arbitrary ANN of three layers: an input layer with two nodes, a *hidden* layer with three nodes and an output layer with two nodes. Given the context of ANNs, a *hidden* layer means that it has no direct interaction with external components of the virtual environment the network is run in; hidden layers are internal and are only used by the network itself [27]. These hidden layers, therefore, represent a latent-space representation of the input data.

FFNs integrate single nodes as presented in Figure 2.2, which in a stacked and layer-wise manner form the complete network of, *e.g.*, Figure 2.3. While the figure omits connection directions and weight annotations for clarity, each connection (*i.e.*, line) represents a learnable scalar weight, directly affecting information flow through the network. Notably, there are no connections between nodes of the same layer, and information flows from left to right: the network is unidirectional.

More specifically, the network in Figure 2.3 represents a densely (or fully) connected network; every node of layer l is connected to every node of adjacent layers $l-1$ and $l+1$.

Consequently, we can express the values of each layer as vectors. For the network shown in Figure 2.3, we see that these vectors will have dimensions of $\mathbf{a}^{(x)} \in \mathbb{R}^2$, $\mathbf{a}^{(h)} \in \mathbb{R}^3$ and $\mathbf{a}^{(\hat{y})} \in \mathbb{R}^2$. Similarly, we can express connections between layers as matrices. Given the architecture of Figure 2.3, we see that the inputs (x_0 and x_1) to hidden (h_0, h_1 and h_2) is a matrix of dimension $\mathbf{W}_{i \mapsto j} \in \mathbb{R}^{3\times 2}$, where each element, $w_{i,j}$, represents the weight connecting x_i with h_j. And likewise for the hidden-to-output weights.

2.3.1 Forward propagation

When the network of Figure 2.3 receives some input values x_0 and x_1, these are fed forward to the hidden layer through a weighted sum of the connections between node x_i and h_j [48], as in Equation 2.1. A forward pass through a single layer can thus be expressed as

$$z_j = \sum_{i=0}^{n} a_i w_{i,j} + b_j \ , \tag{2.2}$$

where z_j denotes the raw output of node j, n is the number of input nodes, a_i is the input value from node i of the previous layer (*e.g.*, x_0), and $w_{i,j}$ is the weight connecting node i (of layer $l-1$) with node j (of layer l).

By following the vector and matrix notation previously defined, we can express this elegantly through

$$\mathbf{z}^{(l+1)} = \mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)} \ , \tag{2.3}$$

where the superscript represents the layer.

Activated values

After calculating z_j, an activation function $\sigma(\cdot)$ is (as mentioned) applied, which introduces non-linearity, allowing the network to learn complex representations of the data [43, 49]. The final value, a_j, of node j is thus given by:

$$a_j = \sigma(z_j) = \sigma(\sum_{i=0}^{n} a_i w_{i,j} + b_j) \ . \tag{2.4}$$

Consequently, forward propagation can be viewed simply as a series of nested input projections onto a latent space.

In practice, each layer typically uses a shared activation function, meaning all pre-activation values (z_j) in any arbitrary layer are processed using the same $\sigma(\cdot)$ but may differ between layers. While using the same activation function for all nodes of a layer is not strictly necessary, it significantly simplifies the (vectorised) forward and backward propagation by ensuring consistent functionality and derivatives per layer:

$$\mathbf{a}^{(l+1)} = \sigma\left(\mathbf{z}^{(l)}\right) = \sigma\left(\mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)}\right) \ . \tag{2.5}$$

While the choice of activation functions is primarily arbitrary and (somewhat) based on empiricism, no one function fits all use cases. Intuitively, the activation function of the output layer is closely related to the problem at hand. When dealing with classification *contra* regression, the choice of the activation function of the output layer matters significantly. With this in mind, a step function

$$\sigma_{\text{out}}(z_j) = \begin{cases} -1 & z_j < 0 \\ 1 & z_j \geq 0 \end{cases} \tag{2.6}$$

would be unsuitable for a regression task. Likewise, a linear function

$$\sigma_{\text{out}}(z_j) = z_j \ , \tag{2.7}$$

suitable for regression would not work well with binary classification.

Expanded network representation

An updated representation of the network of Figure 2.3 can thus be presented, seen in Figure 2.4, with the pre- and post-activation values used through Equations (2.2) and (2.4) included.

When processing inputs through a trained network, these in-between values are redundant (and can, therefore, be discarded), as we are only after the final output. During training, however, all intermediate values are needed.

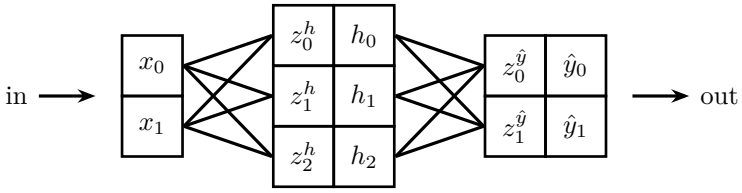


Figure 2.4: A simple ANN of three layers with the pre-activations included.

2.3.2 Backward propagation of error

Through gradient-based optimisation, an ANN updates its weights according to the predicted and expected output. By expanding Figure 2.4 to include the expected values y_0 and y_1, we can compute the gradient of the error, E, between the predicted and actual values based on some objective function, their placeholders seen in Figure 2.5 denoted as g_0 and g_1. These gradients are then backpropagated through the network via the chain rule, modifying each weight to predict better the expected output based on the given input [48, 50, 51].

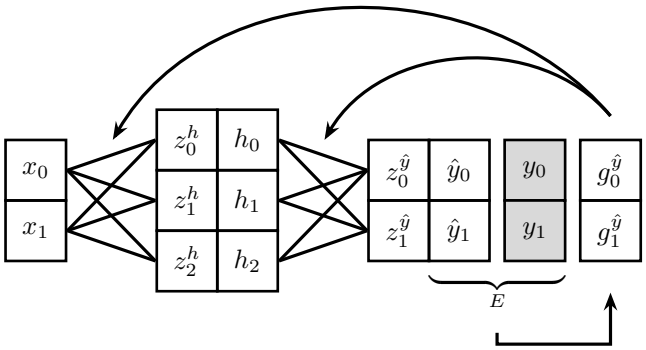


Figure 2.5: Error computation and backpropagation of predicted values $\hat{y}_j$ with respect to the expected values y_j.

Gradients

To correctly update the network weights, gradient descent is commonly used [43, 51–53]. By iteratively navigating the gradient landscape of some objective function, the ANN learns to minimise the error of its predictions.

By assuming an mean squared error (MSE) objective function

$$E = \frac{1}{n}\sum_{j=0}^{n}(\hat{y}_j - y_j)^2 \ , \tag{2.8}$$

where n represents the number of outputs, we can derive the gradient for output j, $g_j^{\hat{y}}$, by differentiating the error of the predicted value, $\hat{y}_j$:

$$g_j^{\hat{y}} = \frac{\partial E}{\partial \hat{y}_j} = \frac{2}{n}\left(\hat{y}_j - y_j\right) \ . \tag{2.9}$$

Chain rule

Focusing on the hidden to output weights, each output node j with the activated value of $\hat{y}_j$ (seen in Figures 2.3 and 2.5) depends on both the activation function, $\sigma_{\text{out}}(\cdot)$ and the value before activation, $z_j^{\hat{y}}$ (Equation (2.4)). Thus, the importance of each weight connecting the hidden- and output neurons, denoted as $w_{i\to j}$, can be found by applying the chain rule:

$$\frac{\partial E}{\partial w_{i\to j}} = \frac{\partial E}{\partial \hat{y}_j} \times \frac{\partial \hat{y}_j}{\partial z_j^{\hat{y}}} \times \frac{\partial z_j^{\hat{y}}}{\partial w_{i\to j}} \ . \tag{2.10}$$

Here, $\frac{\partial \hat{y}_j}{\partial z_j^{\hat{y}}}$ is the derivative of the activation function at the output: $\sigma'(z_j^{\hat{y}})$, and $\frac{\partial z_j^{\hat{y}}}{\partial w_{i\to j}}$ is the activated value in the prior layer: h_i. Thus, Equation (2.10) can be rewritten as

$$\frac{\partial E}{\partial w_{i\to j}} = g_j^{\hat{y}} \times \sigma'(z_j^{\hat{y}}) \times h_i \ . \tag{2.11}$$

To further propagate the output error through preceding layers of the network, we also compute the *input* gradient, where *input* means the gradients for the values of layer i. For our example of hidden to outputs, the *input* gradient would therefore be $\frac{\partial E}{\partial h_i}$, with its value calculated as

$$\begin{aligned} g_i^h = \frac{\partial E}{\partial h_i} &= \sum_j \frac{\partial E}{\partial \hat{y}_j} \times \frac{\partial \hat{y}_j}{\partial z_j^{\hat{y}}} \times \frac{\partial z_j^{\hat{y}}}{\partial h_i} \\ &= \sum_j g_j^{\hat{y}} \times \sigma'(z_j^{\hat{y}}) \times w_{i\to j} \ . \end{aligned} \tag{2.12}$$

Similarly, the *input*- and weight gradients of any layer may be found by propagating the output error backwards through the network, replacing all variables of Equations (2.10), (2.11) and (2.12) with the respective preceding values (*i.e.*, $\hat{y}_j \Rightarrow h_j$, *et cetera*). Intuitively, this process needs to be done from the last layer to the first, *i.e.*, backwards, as it is the output gradient that is propagated and what determines the preceding input gradient(s) (Equation (2.12)).

Optimiser

The actual weight update, given its respective gradient, is after that performed through a chosen optimisation technique (*optim.*):

$$w_{i\to j} = \mathit{optim.}\left(w_{i\to j}, \frac{\partial E}{\partial w_{i\to j}}\right) \ . \tag{2.13}$$

Such optimisers may vary significantly and include stochastic gradient descent (SGD) [54], adaptive moment estimation (Adam) [55] and root mean squared propagation (RMSprop) [56], among others, with SGD being the simplest:

$$optim.(w, \frac{\partial E}{\partial w}) = w - \eta \frac{\partial E}{\partial w} \ , \tag{2.14}$$

where η is an arbitrary scalar referred to as the learning rate.

Whereas activation functions may differ between layers, the optimiser is generally shared across the entire network.

Vectorised notation

Similarly to forward propagation, backward propagation can be expressed through vectorised notation. However, instead of duplicating all equations, we omit these, focusing purely on single-value notation.

2.3.3 Convolutional neural networks

In the simplest case, an FFN consists of densely connected nodes, as previously described. However, for high-dimensional inputs, such as images and videos, dense layers require increasing computational power and (possibly) positional dependency loss, as all nodes of layer l are connected to all nodes of layer $l+1$. To combat these issues, the convolutional neural network consisting of convolutional layers [57] was created, which convolves the input with a kernel yielding the output, which significantly reduces the number of trainable parameters, as the kernel consists of smaller dimensions compared with the input, and at the same time (without padding the input beforehand) produces a smaller output consequently reducing the subsequent trainable parameters.

As presented in Figure 2.6, convolutional layers are often used with dense layers, although pure convolutional neural networks exist (*e.g.*, for semantic segmentation [58], U-net [59] and auto-encoder [60] architectures *et cetera* where the output is also multi-dimensional), they are pretty uncommon.

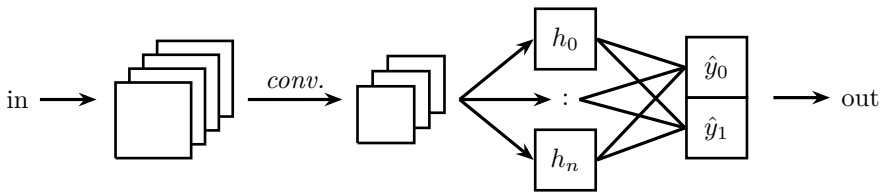


Figure 2.6: A simple CNN with one convolutional layer (marked *conv.*).

When converting between higher-dimensional (*e.g.*, convolutional) layers and one-dimensional dense layers, flattening the outputs of the former is necessary: sequentially chaining rows of the tensor to produce a one-dimensional vector. *I.e.*, simply reshaping the values into a *flat* representation.

Max-pooling layers are commonly used after convolutional layers to reduce the complexity of high-dimensional representations before flattening and processing through dense layers. This technique works by sliding a fixed-size window across the input (typically two-dimensional: for multichannel inputs, each channel is processed individually and stacked), extracting the maximum value within each window. As illustrated in Figure 2.7, a 2×2 filter with a stride of 2 effectively halves the input dimension.

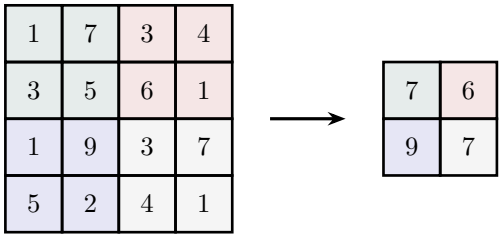


Figure 2.7: A simple max-pool layer with a 2×2 window and stride of 2. The input values (left) are arbitrary.

Starting with the initial window covering the green area of the input of Figure 2.7, the maximum value is extracted as the output. The window then shifts according to the specified stride, moving two indices to the right (*i.e.*, to the red region) in our example. When reaching the edge, the window repositions to the left and moves down by the stride amount (*i.e.*, to the blue region), continuing the process.

Backpropagation of both convolutional and max-pooling layers is omitted here. However, their implementations are found in the accompanying framework by Lavik [4].

2.4 Recurrent neural networks

FFNs observe a single state (the input) and linearly project these values to produce a single output, lacking the mechanism to consider past information[1] or iterate on some latent space representations[2]. Consequently, the linear processing of data in FFNs leads to issues concerning temporal dependencies found in *e.g.*, sequential tasks. Although it is possible to encode positional information directly into the input vector (which is incorporated into SOTA LLMs [61]), such an approach might not be suitable for a wide array of problems [30].

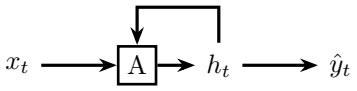
To address the challenges of positional dependency, Elman [30] (based on prior work by Jordan [31]) proposed the RNN. RNNs incorporates a state vector containing the context of prior information when processing new inputs, consequently

[1] Assuming this is not explicitly included in the input.

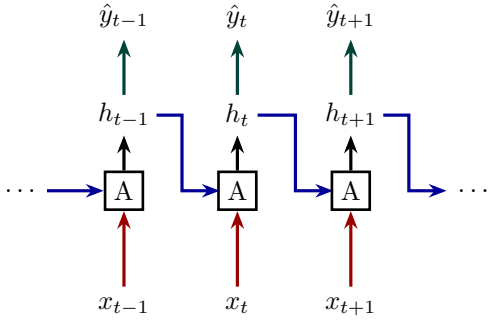
[2] Of course, a deeper network would imply implicit latent-space iteration, enhancing certain features, *et cetera*. But, here, we are referring to a fixed-size network and its ability to iterate on a given input without the need for a deep network.

capturing temporal dependence. Thus, the design of the initial RNNs provided a solution to the limitations of FFNs in sequential tasks by eliminating the need for explicit temporal definition.

The basic structure of an RNN is illustrated through Figure 2.8, where A represents an ANN, and h_t represents the state vector containing the *memory*.



(a) Compact representation where each arrow represents a unique set of weights between the respective connected elements. The arrow into h represents the output layer of A.



(b) Unwrapped representation where each arrow represents a set of weights between the respective connected elements. The arrow into h represents the output layer of A. Weights are identical for the different time lags, with equal weights shaded the same colour.

Figure 2.8: A simple RNN.

Shared weights

Notably, the network weights are identical across the different time lags. A property that is intuitive when considering the compact representation (*i.e.*, Figure 2.8a) compared to the unwrapped network representation (*i.e.*, Figure 2.8b). To highlight this property, equal weights are shaded the same colour in the unwrapped representation.

A direct consequence of the weight sharing property is the reduced memory footprint of the network, compared to a deep network of equivalent size as the unwrapped RNN.

Temporal backward propagation

An apparent problem regarding RNNs is that the output at time t depends on all previous outputs $[0..t]$ through the hidden state of time $t-1$. To backpropagate the error, we need to unwrap the structure and similarly start at the end as for FFNs, accounting for all preceding layers [33]. Through Equation (2.15), we see how the (arbitrarily chosen) weight gradient for the mapping between h and A is found:

$$\begin{aligned}\frac{\partial E_t}{\partial W_{h\mapsto A}} &= \frac{\partial E_t}{\partial \hat{y}_t} \times \frac{\partial \hat{y}_t}{\partial h_t} \times \sum_{k=1}^{t} \left(\frac{\partial h_t}{\partial h_k} \times \frac{\partial h_k}{\partial W_{h\mapsto A}} \right) \\ &= \frac{\partial E_t}{\partial \hat{y}_t} \times \frac{\partial \hat{y}_t}{\partial h_t} \times \sum_{k=1}^{t} \left(\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \times \frac{\partial h_k}{\partial W_{h\mapsto A}} \right) ,\end{aligned} \tag{2.15}$$

with a similar approach for the other gradients. In this way, we ensure all connections are coupled at all times.

2.5 Skip connections

A simple trick allowing deep networks to retain context in progressively more abstract latent representations is achieved by introducing skip connections (also referred to as residual connections) [62]. Imagining a network composed of a subsystem[3] $\boxed{\text{A}}$, a bypassing skip connection would imply that the output of this subsystem is being combined with the input to the subsystem, as seen in Figure 2.9. In the figure, the combining is represented as *op.*, which typically denotes an element-wise addition [62], but could be exchanged with any function (*e.g.*, concatenation, multiplication *et cetera*).

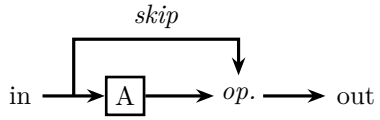


Figure 2.9: A simple skip connection.

Consequently, information is both bypassed and processed through the learnable parameters (as presented in Figure 2.3), and the final output can be expressed through

$$\text{out} = \textit{op.}\left(\text{A}(\text{in}), \text{in}\right) , \tag{2.16}$$

where `A(in)` denotes the output of subsystem $\boxed{\text{A}}$. In this manner, we allow the latent representation (*i.e.*, the output of $\boxed{\text{A}}$) to retain the context of the input.

Through Equation (2.16) it becomes apparent that the dimensions of `A(in)` and `in` must be equal when *op.* is assumed to be the element-wise sum. However, this may easily be bypassed by modifying the skip connection to be a (learnable) projection to assert their equality [62]. That is, the skip connection can be considered a layer itself. When mentioning skip connections, however, we assume these to be identity mappings.

[3] When mentioning subsystems of ANNs, we can think of these as either a single layer of the network or an entire network. That is, a valid subsystem could, therefore, for instance, be the network seen in Figure 2.3.

While skip connections might not be beneficial for shallow networks (*i.e.*, networks of few layers), it has proved extremely valuable for deep networks, especially applied to problems regarding object detection [5, 59, 62, 63], mitigating gradient-related problems ([64]).

Backward propagation

Assuming a skip connection where *op.* represents an element-wise addition

$$\text{out} = \text{A}(\text{in}) + \text{in} \ , \tag{2.17}$$

the backward propagation of error is quite simple. Intuitively, as the input to $\boxed{\text{A}}$ directly influences the output, the gradient must account for this. Given the chain rule, we can formulate the gradient flow through the connection as

$$\begin{aligned} \frac{\partial E}{\partial \text{in}} &= \frac{\partial E}{\partial \text{out}} \times \frac{\partial(\text{A}(\text{in}) + \text{in})}{\partial \text{in}} \\ &= \frac{\partial E}{\partial \text{out}} \times \left(\frac{\partial \text{A}(\text{in})}{\partial \text{in}} + \frac{\partial \text{in}}{\partial \text{in}} \right) \\ &= \frac{\partial E}{\partial \text{out}} \times \frac{\partial \text{A}(\text{in})}{\partial \text{in}} + \frac{\partial E}{\partial \text{out}} \times 1 \ , \end{aligned} \tag{2.18}$$

where $\frac{\partial E}{\partial \text{out}}$ represents the output gradient.

Through Equation (2.18), it becomes apparent that the input gradient is simply the sum of the gradient of the skipped subsystem and the output gradient. In this manner, skip connections prove directly advantageous for deeper networks experiencing gradient vanishing [62].

3 Methods

Drawing inspiration from the high number of feedback connections in biological neural networks, we present a modification to the ANN architecture exhibiting (at an abstract scale) this behaviour. While our approach is certainly not biologically plausible, our primary motivation is its aforementioned biological counterpart.

3.1 Feedback connections in artificial neural networks

Building on ANNs by combining finite recurrence and skip connections, we can define what we will hereafter call feedback connections. As the name hints, a feedback connection is a pathway between a temporally subsequent output vector looped back as input to a previous layer. In this manner, specific layers are reused, processing values more than once. Consequently, the terms *feedback* and *loop* are used interchangeably. To get a basic grasp of feedback connections in ANNs, we consider an abstract network presented through Figure 3.1, where A and B are subsystems of the network, which represents one or more layers of an FFN; subsystems are either a single layer or entire networks of multiple layers. These subsystems thus represent two individual projections of some input to an output.

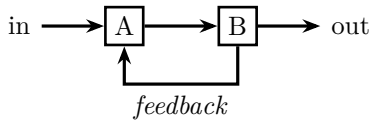


Figure 3.1: A simple feedback connection in an abstracted network with arbitrarily two subsystems.

Whereas RNNs repeatedly processes new inputs along with its recurring hidden state yielding outputs (Figure 2.8b), a feedback connection is internal to the network, directly modifying information flow in a non-recurring fashion. Non-recurring in the sense of not being autoregressive or relying on subsequent inputs, but recurrent concerning layer reuse; once the values of a feedback connection have been fed back, this connection is *dead* when the fed back values again reach the feedback layer:

a recursion of (in this case[1]) depth 1. In this sense, networks employing feedback connections do not require additional inputs or yield outputs per temporal step, as seen in RNNs. However, concerning the hidden state of the latter, we compare this to that of the fed back latent representation (*i.e.*, the output of [B] in Figure 3.1 which is fed back to [A]).

In the following subsections, we will describe how the simple feedback connection of Figure 3.1 leads to complicated behaviour in ANNs. By first building an intuition behind the forward pass of a feedback loop, we can later expand this to incorporate logic regarding backward propagation.

3.1.1 Connections

Before we progress, however, we should clarify some aspects of the feedback connection, namely how the subsequent output is looped back. In the simplest case, the connection (marked *feedback* in Figure 3.1) resembles a direct connection. *I.e.*, the output of [B] serves as the sole input to [A] when it is being fed back.

As we will see later, this connection could also represent a skip connection, retaining the original input by combining it with the subsequent latent representation. In these cases, we assume the connection to be an identity mapping, followed by an (*op.* of) element-wise addition. Consequently, these do not contribute any learnable parameters. While it is possible to define feedback connections as a separate learnable projection of the network (as with skip connections), we purely investigate the base case where it is not. Additionally, if the feedback connection is assumed to be learnable, this would complicate the backward pass further and would be inherently unsuitable for especially (what we call) COMBINE, which we will define and explore later.

Matching shapes

Consequently, we can obtain from Figure 3.1 some technicalities related to the input and output shapes of subsystems [A] and [B]. By assuming feedback connections to be direct or identity mappings with *op.* as the element-wise sum, we gather that the input to [A] must match that of the output of [B], to accommodate the feedback connection.

3.1.2 Forward propagation

In the context of feedback connections, as the one of Figure 3.1, we need to determine how the input (`in`) should be projected through the subsystems [A] and [B]. By unwrapping the network, we identify three primary methods, depicted in Figures 3.2, 3.3 and 3.4.

[1]The recursion depth is in the accompanying code set as a variable parameter, for the user to determine.

Linear unwrap

The most straightforward approach to unwrapping the feedback connection of Figure 3.1, presented in Figure 3.2, involves stretching the network; the feedback connection is unwrapped linearly. The resulting network thus becomes a deeper version of its origin (*i.e.*, Figure 3.1).

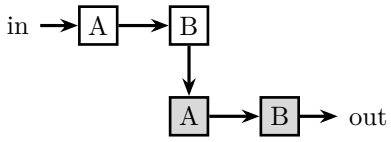


Figure 3.2: Linear unwrap.

Consequently, the network can further process its latent representation (*i.e.*, the output of B) by feeding this back as input to the network (*i.e.*, to A), in this way reusing existing weights present inside the respective subsections.

Unwrap with input skip connection

Additionally, we have the possibility of incorporating a skip connection from the initial input (*i.e.*, `in`) to the fed-back input (*i.e.*, the output of B), seen in Figure 3.3.

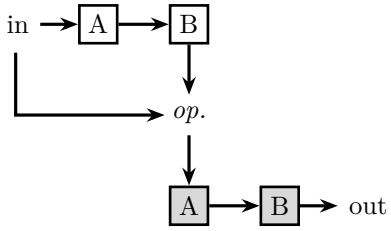


Figure 3.3: Unwrap with input skip connection.

In this manner, mimicking the U-net architecture [59] by producing a richer latent representation before further processing. Through this configuration, the network may enhance or diminish certain features of the input data by directly combining it with the processed representation. Not unlike how long short-term memory (LSTM) models [65] include specific gating mechanisms.

Consequently, we hypothetically force the model to rely more on the input data, as opposed to the linear unwrapping of Figure 3.2, where the second iteration purely depends on the latent representation of the input.

With the inclusion of an input skip connection (as seen in Figure 3.3) where the operation directly modifies the shapes[2], our assumption of matching shapes is further complicated. In these cases, the connection between the operation of the input skip connection and A *must* represent some projection. As a result, we do not consider these architectures due to their complications of the backward pass.

[2]Such as concatenation.

Unwrap with output skip connection

Much like the unwrap with input skip connections, seen in Figure 3.3, we can combine respective iteration outputs, presented in Figure 3.4.

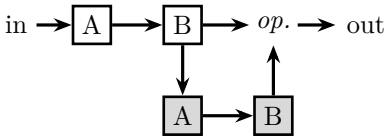


Figure 3.4: Unwrap with output skip connection.

Here, instead of retaining the input upon further latent processing, we postulate that the initial output (*i.e.*, that of [B]) leads to a higher contextualised final output (*i.e.*, that of [B]).

Similarly to networks with input skip connections (Figure 3.3), including output skip connections forces the final network output to rely equally[3] on each iteration's output.

Other configurations

Additionally, the subsystems themselves could contain internal skip connections; if a subsystem is a network itself (*i.e.*, not a single layer), this subnetwork could incorporate skip connections in the regular sense, as described in Section 2.5. However, when discussing feedback connections in this chapter, we do not include these to clarify better and present the feedback property. These are, however, easily incorporated through the accompanying framework [4].

In Figure 3.5, an additional skip connection is introduced, connecting the original input representation with the second iteration output (*i.e.*, the output of [B]) before further processing, preserving both the initial input and the further abstracted representation. In this manner, the data flow is separated into two paths; one path goes through the feedback section, with the other directly bypassing it. Interestingly, implementations of these connections allow us to probe the importance of the two pathways, as we will discuss later.

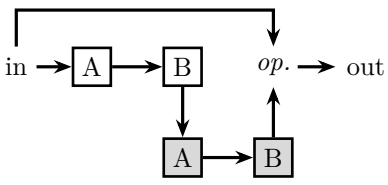


Figure 3.5: Unwrap with bypassing skip connection.

It is worth noting that the aforementioned approaches of unwrapping the feedback loop are non-exhaustive. Depending on the number of subsystems in the feed-

[3]Depending on the combination function, of course, but as we assume element-wise addition operations (*op.*), each element contributes with equal weighting.

back loop, a greater number of skip connections can be included. Even for our simple case containing only two subsystems, our presentation of a few skip connections does not handle all possible scenarios, as the mentioned configurations may be combined to form new ones.

Consequently, when experimenting with feedback architectures, we include only those with no skip connections (Figure 3.2) and those with bypassing skip connections (Figure 3.5). However, all architecture types have been implemented in the accompanying framework [4], with a simple boolean flag to toggle either input or output skip connections.

Weight sharing

Importantly, as in RNNs, the weights of respective unwrapped feedback loop subsystems are shared. In Figures 3.2, 3.3, 3.4 and 3.5, the unwrapped nodes are shaded: A and B, signifying that they are clones of their original subsystem. That is, A and A are identical copies of one another, *et cetera*.

This property is evident when inspecting the compact representation of Figure 3.1, similarly to for RNNs (Figure 2.8). In this way, we can reduce the total number of weights of an ANNs and, in turn, potentially gain a richer and more performant network by mimicking deeper networks through layer reuse.

3.1.3 Backward propagation

Considering the feedback connection illustrated in Figure 3.1, we must determine how to propagate the output gradient through the loop and the rest of the network. And since it is crucial to couple the weights, we need to ensure that respective subsystems (*e.g.*, A and A) match after backpropagation.

Basing backpropagation and weights updates on FFNs rather than RNNs (which is done by [5, 6]), two main approaches present themselves as a consequence of subsystem coupling:

COMBINE	Combine the pre- and post-activations of respective subsystems before backpropagation, ensuring identical respective subsystem weights at all times.
COUPLE	Treat the unwrapped representation as a deeper network (as is done in the forward pass) during backpropagation, coupling the respective sets of weights after updates.

This nomenclature was chosen to reflect their differences during backpropagation. The following subsections will detail these approaches, focusing on the simple linearly unwrapped case shown in Figure 3.2.

As mentioned, A and B represent subsystems of an ANN. For backward propagation of an FFN (discussed in Sections 2.3.1 and 2.3.2), the pre- and post-activated

values of the layers within these subsystems are needed. These pre- and post-activations will be referred to as *values*.

COMBINE

Rearranging the unwrapped loop slightly and aligning the respective subsystems vertically to enhance intuition better, as seen in Figure 3.6, we can connect *values* of the shaded *forward* section, resulting in a combined representation (the bottom *backward* part). The final output of the forward pass generates an error (as explained in Section 2.3.2), E, which is then backpropagated through this combined representation of the *values*, also included in the figure.

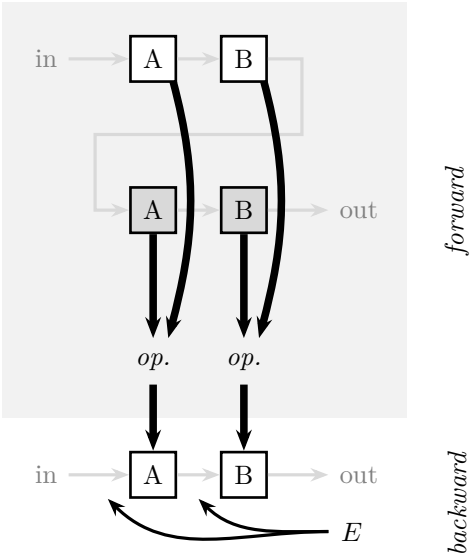


Figure 3.6: Unwrapped feedback loop for COMBINE. The shaded region represents the forward pass, and the unshaded part represents the backward pass, with combined *values* (shown by thick arrows) from the forward pass.

Consequently, when backpropagation is done using the combined *values*, we directly ensure that the weights are coupled for the subsequent forward pass. After the error has been backpropagated through the combined *values*, the network can again be unwrapped for new inputs, thereby repeating the process illustrated in Figure 3.2 continuously during training, similarly to how RNNs operate.

To further clarify COMBINE, we replace the abstractions of Figure 3.6 by exchanging the subsystems with the network of Figure 2.4. We then introduce a feedback connection from the output to the input, as shown in Figure 3.7.

Similar to our abstracted examples, the second iteration through the network is shaded. To better enhance identical weights, we colour-code these, similarly to the RNN of Figure 2.8b. That is, each set of weights of the same colour is equal. Additionally, we superscript each node with its iteration count, signifying that the *values* of each iteration are unique.

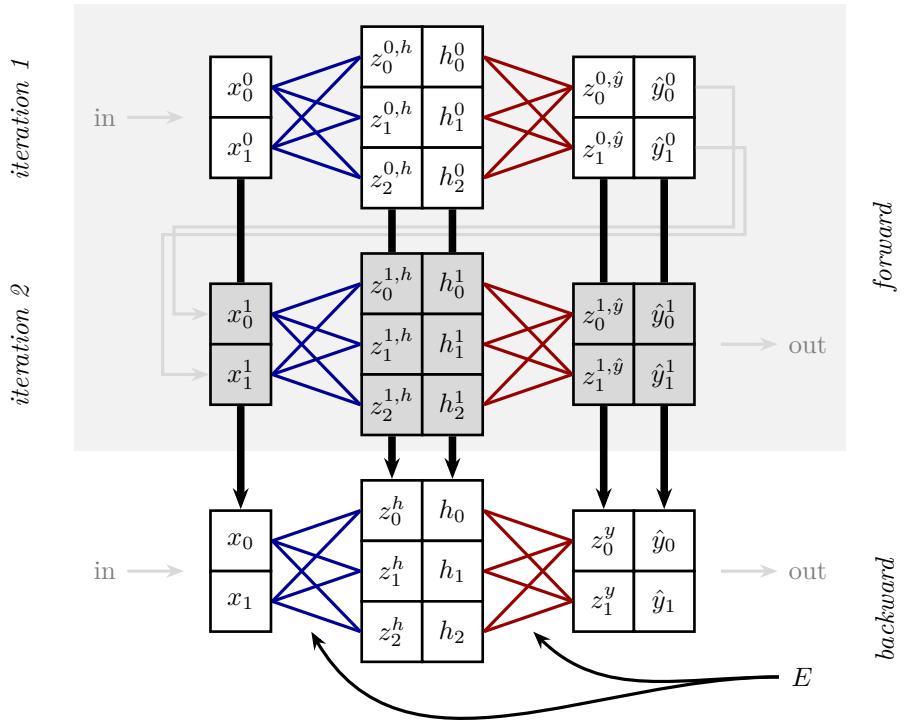


Figure 3.7: COMBINE for a network. The shaded region represents the forward pass, and the unshaded part the backward pass with the combined *values* (marked by thick arrows, indicating their combination, as in Figure 3.6).

Based on the *values* in Figure 3.7 we can define their combination as

$$v_i = op.\left(v_i^0, v_i^1, \ldots, v_i^j\right) \quad \text{for all looped iterations } j \in [0..m]\,, \tag{3.1}$$

where v_i represents the final values, m is the number of loops, and v_i^j is the individual value per unwrapped layer. The operation (*op.*) could be by any chosen operator, such as the sum ($v_i = \sum_{j=0}^{m} v_i^j$) or the mean ($v_i = \frac{1}{m}\sum_{j=0}^{m} v_i^j$), *et cetera*. For our example network of Figure 3.7, assuming *op.* to be averaging, we can compute h_0 as $\frac{1}{2}\left(h_0^0 + h_0^1\right)$.

These combined *values* then serve as the true *values* of the compacted feedback network and are directly used when backpropagating.

COMBINE automatically handles weight coupling and minimises computational cost, making it ideal for large feedback loops or extensive loop counts. Compared to the equivalent regular network (*i.e.*, Figure 2.4), the time complexity of weight updates remains unaffected by the loop count m, as the backward propagation is, as mentioned, applied to the compacted combined *values*. On the other hand, the forward pass increases time complexity as a function of both iteration count, m, and the feedback size (*i.e.*, the number of parameters used for repeating projections).

While skip connections can be included in the forward pass, no intuitive way of

backpropagating through time for the combined *values* presents itself. With this in mind, the resulting backpropagation could lead to unwanted results if skip connections are present in the forward pass.

COUPLE

While COMBINE reduces costs, it may affect interpretability and introduce noise due to the intertwining of original and looping values during backpropagation. Conversely, COUPLE treats the unwrapped representation as a regular deep network, as presented in Figure 3.8, with the condition of coupled weights for respective subsystems. Consequently, as we backpropagate the error through the unwrapped representation, training a network with this approach is comparable to an equivalent regular deep network, with the addition of weight coupling. However, the resulting trained network is of significantly smaller size (as opposed to its deep equivalence), as subsystems are identical and reused.

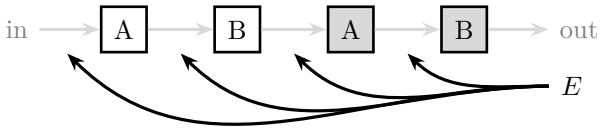


Figure 3.8: COUPLE for an abstracted network.

Similarly to COMBINE, we can substitute subsystems of Figure 3.8 with the network of Figure 2.4, presented through Figure 3.9.

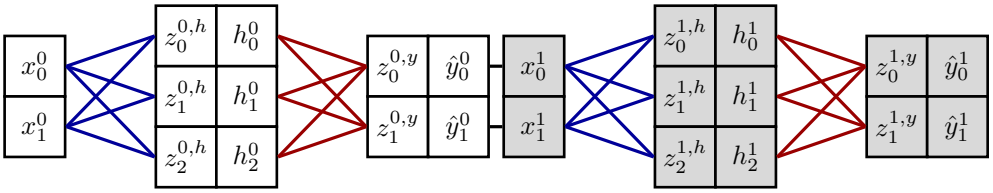


Figure 3.9: COUPLE for a network.

Since the unwrapped representation is used directly, gradient accumulation related to skip connections, as explained in Section 2.5, is easy to include. Consequently, including skip connections to the architecture poses no difficulties for backpropagation in COUPLE.

Once the error has been backpropagated, we must couple the subsystems presented in Section 2.3.2. That is, given the unwrapped (updated) network, as the one of Figure 3.9, we must ensure that all weights connecting x_i^0 with $z_j^{0,h}$ are equal to those connecting x_i^1 with $z_j^{1,h}$, and so on. That is, modifying all sets of equally coloured weights of Figure 3.9 to match. Here, we can freely choose how this is handled, with the constraints of ensuring stability in training. With this in mind, the

most straightforward approach would be to average across the weights of respective subsystems

$$W_{a,b} = \frac{1}{m} \sum_{i=0}^{m} W_{a,b}^{i} \, , \tag{3.2}$$

where $W_{a,b}$ represents the coupled weights connecting layer a with b (*e.g.*, the blue connections). $W_{a,b}^{i}$ represents the updated weights for subsystem i (*e.g.*, all the connections from x_j^1 to $h_k^{1,h}$).

3.2 Deep learning framework

While numerous DL frameworks exist, most rely on automatic gradient computation. Since our feedback approach COMBINE backpropagates using the combined *values* of the forward pass, automatic gradients are unwanted, making standard frameworks unsuitable. Consequently, we developed the DL framework `neurons` [4] from scratch, adding an extra challenge beyond the general thesis. The framework, with documentation, tests and examples, is all made open-source and found in the repository by Lavik [4].

Programming language

Despite a background in Python [66], we chose Rust [67] as the programming language behind `neurons` due to its general performance improvement compared to the former. Although `PyTorch` [68], `TensorFlow` [69], and similar frameworks are accessible through Python, they primarily serve as wrappers for more efficient, low-level languages such as C++. While our choice of Rust was arbitrary, as any efficient language would suffice, its memory safety and somewhat similar syntax to Python were driving factors. Consequently, learning Rust, developing a thorough from-scratch DL framework, and implementing novel feedback connections led to an enriching and enjoyable learning experience.

Difficulties

With the complete framework only including a single dependency (`Rayon` [70], a concurrency library), the magnitude of general implementation was quite challenging. Especially considering my limited prior experience with Rust.

High-dimensional convolutional layers were particularly complex to implement. Balancing code efficiency with flexibility required significant effort, with further room for improvement. Consequently, without the help of AI assistants [71–75], the present state of `neurons` would not be at its current level.

3.2.1 Flexibility

Throughout development, framework flexibility and a seamless user experience were prioritised. With a flexible design, the application programming interface of `neurons` somewhat resembles that of `TensorFlow` [69]. Namely, we centre ANNs and its functionality around a network object, allowing us to dynamically modify its layers and other characteristics. This similarity may be seen through the code examples of Section 3.2.5 or in the examples directory of the framework.

Tensors

While it is possible to define generic functions concerning typesetting in Rust, this adds a layer of abstraction to the code. Therefore, we choose to implement everything with fixed types. Consequently, a custom *Tensor* structure is needed to avoid duplicated function definitions, as the type of a one-dimensional vector is unequal to that of a two-dimensional vector, *et cetera*. Therefore, the foundation of our framework becomes this *Tensor* structure, which contains all[4] values of our networks, providing a single interface for all operations. This way, function definitions are simplified as we specify inputs (and outputs) to be *Tensors*, thus complying with Rust's strict typesetting.

Layers

When defining a network, the user specifies the layers it is comprised of sequentially. The relevant shapes are automatically inferred based on the preceding layer. When defining a new layer, one specifies the number of nodes (for dense layers) or the kernel shape (for convolutional-like layers). For the latter layer types, output shapes are additionally inferred.

Conversions between layer types (*e.g.*, between dense and convolution) are also supported. Here, the network automatically applies flattening, or oppositely, converting one-dimensional *Tensors* to higher-dimensionality, yielding an error for impossible conversions; for conversions between one-dimensional and three-dimensional data, the closest possibilities are suggested, and so on. For example, suppose a network is defined sequentially with a dense layer of 99 nodes and a convolutional layer thereafter. In that case, the network yields an error suggesting the former dense layer should instead be of 100 nodes, as this can be (losslessly) reshaped to $1 \times 10 \times 10$, instead of the defined 99 nodes.

In addition to the dense and convolutional layers, `neurons` implement deconvolutional, max-pooling, and feedback layers.

[4] That is, all values where functionalities related to these are dynamic based on shapes. Consequently, single-valued attributes (such as the learning rate, *et cetera*) are not necessarily stored in *Tensors*.

Functions

All the aforementioned layers include a specific activation function, which is included when defining a new layer. The following activations are implemented:

- Linear
- Sigmoid
- Tanh
- Rectified linear unit (ReLU)
- Leaky rectified linear unit (LeakyReLU)
- Softmax

Additionally, the entire network has a single objective function which computes the output loss and gradient, along with an optimisation technique for weight updates. The implemented objective functions and optimisation algorithms are:

Objective functions

- Absolute error (AE)
- Mean absolute error (MAE)
- Mean squared error (MSE)
- Root mean squared error (RMSE)
- Cross-entropy
- Binary cross-entropy
- Kullback-Leibler divergence (KL-divergence)

Optimisation techniques

- Stochastic gradient descent (SGD)
- Stochastic gradient descent with momentum (SGDM)
- Adaptive moment estimation (Adam)
- Adaptive moment estimation with weight decay (AdamW)
- Root mean squared propagation (RMSprop)

3.2.2 Trivial implementation

While `neurons` include a diverse range of functionalities, it lacks certain features found in its well-established inspirations. As mentioned, the framework is built entirely from scratch, apart from the aforementioned concurrency library, `Rayon` [70]. This thesis investigates feedback connections in FFNs, and more complex architectures, such as LSTMs and LLMs, have consequently not been implemented. However, it is worth mentioning that these functionalities could be included in the future. In other words, `neurons` encompass most functionalities related to FFNs.

Validation of correctness

From-scratch implementations are prone to bugs and errors. Therefore, to ensure the trustworthiness of `neurons`, comparisons were made with identical networks implemented in `PyTorch` [68]. These scripts are included in the documentation directory of `neurons`.

Additionally, every module[5] of `neurons` includes thorough unit tests of its functionalities, ensuring robust and reliable performance. Where relevant, these unit tests include comments pointing to their equivalent `PyTorch` [68] implementation.

Concurrency

To fully utilise the speed of Rust, we employ multithreaded mini-batched learning. In other words, we iterate through all samples of a mini-batch in parallel, thereby increasing the total runtime speed in proportion to the specified mini-batch size[6]. At the same time, employing mini-batched training greatly improves stability in learning [76], as weight updates are aggregated and executed once per mini-batch as opposed to per-sample updates[7].

3.2.3 Network

The network module combines all previously mentioned modules of the framework seamlessly. Through this, the individually defined layers are stored, with their respective activation functions, *et cetera*. Following object-oriented programming principles, the `Network` structure (where *structure* is used interchangeably with *object*) then applies the relevant functionality to its attributes, leading to a seamless user experience. A simplified `Network` structure is defined in Listing 3.1.

Listing 3.1: Base network structure.

```
pub struct Network {
    pub layers: Vec<Layer>,
}
```

In the network object (Listing 3.1), we assume the attribute `layers` to hold an ordered array of the individual layers (enumerated through `Layer`) of the network. Picturing a network of three layers, this object would resemble that of Listing 3.2,

Listing 3.2: Example (abstracted) network with three layers.

```
let mut network = Network::create();
network.layers = vec![ in, hidden, out ];
```

where we omit the properties of the individual layers along with other parts of the network for clarity. Additionally, we abstract the contents of the `layers` attribute. The presented Listings 3.1 and 3.2 thus differ from the actual implementation, here simplified to enhance intuition.

[5] With the complete framework being relatively large, it is divided into separate *modules*, which in simple terms may be thought of as files. Consequently, each aspect of the framework is neatly separated, with differing functionalities separated.

[6] Of course, being highly dependent on the available hardware it is being run on.

[7] If the mini-batch is set equal to 1, implementation is effectively identical to per-sample updates.

Forward function

Even though most implementation is trivial, we describe the network-wide forward function to expand this later to incorporate feedback connections. For the forward pass of our network, we express a simplified forward function, presented in Algorithm 1. The forward pass sequentially propagates the input forward through the network's layers, simultaneously storing the *values* of each step for the subsequent iteration and a potential backward pass.

Algorithm 1 Base forward function of an FFN, assuming LAYER(x) computes the respective forward step of the layer with the current `input` representation, `x`.

```
procedure FORWARD(input)
    preactivated ← [ ]
    activated ← [ input ]
    for i in 0..self.layers.length do
        x ← activated.last
        LAYER ← self.layers[i]
        pre, post ← LAYER(x)
        preactivated.next ← pre
        activated.next ← post
    end for
    return preactivated, activated
end procedure
```

3.2.4 Feedback connections

Similarly, as we distinguish the backward propagation of errors through feedback connections into two approaches, their implementations are also separated.

COMBINE operates on a compact network representation, similar to skip connections, albeit in a reverse manner. Consequently, we define feedback connections of this approach through network attributes (as is done for skip connections), specifying layer indices and an input combination flag[8]. In contrast, COUPLE treats the network as a deep, unwrapped structure and is implemented as a separate framework module.

COMBINE

Unlike COUPLE, which requires unwrapping the network through subsystem cloning, COMBINE iterates through subsystems multiple times, thus maintaining a shallow-depth structure. The COMBINE approach thus maintains a relatively low memory footprint by using the compact representation for both forward and backward prop-

[8] See unwrap configurations of Subsection 3.1.2.

agation (as in Figure 3.1). Consequently, this footprint does not scale proportionally to the unwrapped depth, as it does for COUPLE.

By drawing inspiration from skip connections [62] in a reverse manner, we extend our base network to incorporate a `loopback` attribute, presented in Listing 3.3. We then look up these mappings between layer indices, stored in `loopbacks`[9], through both the forward and backwards functions of the network.

Listing 3.3: Network structure extended with loopbacks.

```
pub struct Network {
    pub layers: Vec<Layer>,
    pub loopbacks: HashMap<usize, usize>,
}
```

Similarly, we can update our specific network object with a feedback connection from the output of layer two (*i.e.*, `out`) to the input of layer zero (*i.e.*, `in`), seen in Listing 3.4.

Listing 3.4: Example (abstracted) network with a specific loopback.

```
let mut network = Network::create();
network.layers = vec![ in, hidden, out ];
network.loopbacks.insert(2, 0);
```

By modifying the forward pass of the network to incorporate the `loopbacks` attribute, we can accommodate feedback connections when forward and backward propagating. The forward pass is first divided into two separate functions. One main `forward` function, Algorithm 3, which loops through the network layers once, and one `step` function, Algorithm 2, which is called inside the main `forward` function to process a range of layers. This is done to distinguish the layer iteration between the main loop and potential feedback loops. For networks without any defined feedback connections, *i.e.*, when `loopbacks` is empty, the forward pass becomes equal to that of Algorithm 1.

The primary forward function checks for feedback connections (Algorithm 3). When encountering a loopback index, it feeds the current activation to a previous layer using the `step` function before continuing forward propagation.

Combining *values* directly in the forward function (Algorithm 3) thus directly incorporates output skip connections, as in Figure 3.4. Consequently, this introduces certain disadvantages that will be examined in Chapter 5. However, this approach offers a significant benefit: it reduces the computational complexity of backward propagation. Since the *values* are combined automatically during the forward pass, we eliminate the need for additional iterations through feedback loops before starting backpropagation. Along with excessive memory usage as a consequence of storing the unwrapped *values*.

[9] Note that we, for clarity, omit the number of iterations and flag for input-to-input connections (Figure 3.3) in this example.

Algorithm 2 Forward propagation of *input* for a given range of layers: $[start, stop\rangle$.

```
procedure STEP(input, start, stop)
    preactivated ← [ ]
    activated ← [ input ]
    for i in start..stop do
        LAYER ← self.layers[i]
        pre, post ← LAYER(activated.last)
        preactivated.next ← pre
        activated.next ← post
    end for
    delete activated.first                ▷ To prevent duplication of input.
    return preactivated, activated
end procedure
```

Algorithm 3 Forward pass with feedback connections of COMBINE combining the *values* through a *combine* function of choice, *e.g.*, the element-wise sum, the element-wise mean, *et cetera*.

```
procedure FORWARD(input)
    preactivated ← [ ]
    activated ← [ input ]
    for i in 0..layers.length do
        x ← activated.last
        pre, post ← STEP(x, i, i + 1)                          ▷ Single.
        preactivated.next ← pre
        activated.next ← post
        if i in self.loopbacks then
            input ← activated.last
            fpre, fpost ← STEP(input, self.loopbacks[i], i + 1)  ▷ Mult.
            combine respective values
        end if
    end for
    return preactivated, activated
end procedure
```

COUPLE

COUPLE provides a more straightforward implementation by handling the network in its unwrapped form (Figures 3.8 and 3.7). This eliminates the (complex) combination of *values* and allows using the standard forward pass algorithm (Algorithm 1).

The unwrapped representation used in COUPLE naturally accommodates skip connections during backpropagation, allowing error gradients to flow through all connections, unlike COMBINE, where skip connections are difficult to incorporate

in the backward pass.

COUPLE follows standard FFN backpropagation, with one addition: weight coupling between duplicated layers. While training requires layer duplication for unwrapped feedback connections, these duplicates can be removed post-training to create a compact final structure, mimicking the forward pass of COMBINE by directly iterating through the base subsystem(s).

By defining COUPLE through a separate framework module, we cleanly isolate its unwrapped layers from the rest of the network. Defining this module as feedback blocks, we implement this similarly to other layer types. Consequently, all functionality related to feedback blocks (namely weight coupling) is separated from the standard FFN functionality, exemplified through Listing 3.5 (similarly to Listing 3.7, with certain attributes omitted for clarity).

Listing 3.5: Base feedback structure.

```
pub struct Feedback {
    layers: Vec<Layer>,
    coupled: Vec<Vec<usize>>,
}
```

In Listing 3.6, we create a new `Feedback` object with arbitrary example values. Here, we see that the layout closely resembles the unwrapped representations seen in Figure 3.2 (and, of course, Figures 3.3, 3.4 and 3.5). The additional attribute, `coupled`, holds the coupled layer indices. In our example, the first sublist of `coupled` states that layers 0, 2 and 4 of `layers` (*i.e.*, `a1`, `a2` and `a3`) are identical and should match after any weight updates.

Listing 3.6: Example (abstracted) feedback layer.

```
let mut feedback = Feedback::create();
feedback.layers = vec![ a1, b1, a2, b2, a3, b3 ];
feedback.coupled = vec![vec![ 0, 2, 4 ], vec![ 1, 3, 5 ]]
```

COUPLE feedback loops effectively function as self-contained networks within the larger FFN structure. The main network interacts with these feedback blocks as with standard layers, simply calling their forward and backward functions and using our example feedback block of Listing 3.6, the input interface is through `a1`, and the output is from `b3`. The internal layers (`b1` through `a3`) are encapsulated within the feedback block and remain hidden from the main FFN. While providing clean interfaces, this encapsulation does introduce one limitation: it prevents skip connections between the main layers of the network and the internal layers of feedback blocks (such as, for instance, connecting directly to `b2`).

As we will see later, feedback block attributes are defined through a wrapper function, *i.e.*, not as in Listing 3.6. Consequently, the user specifies the base layers (*e.g.*, `a` and `b`) and the number of feedback iterations (*e.g.*, `3`). The network then automatically unwraps these base layers concerning the iteration count and generates the coupled attribute (which we manually did in Listing 3.6).

3.2.5 Examples

While the described implementation of COMBINE and COUPLE is somewhat simplified here, they capture their essence, with the actual implementation available through [4].

As mentioned, `neurons` is designed to be highly flexible and user-friendly. The framework supports various implementations, including our two novel feedback methods. While the repository contains numerous examples, we briefly demonstrate how to integrate these feedback mechanisms into networks.

Network definition

Before expressing feedback definition, we must describe how the network is created. By importing the relevant modules, we then store the creation of a network with a given input shape to a (mutable) variable, as seen in Listing 3.7.

Listing 3.7: Network creation.

```
// 1. Import the `network` and `tensor` modules.
use neurons::{network, tensor};

// 2. Define the input shape.
//    Here: a one-dimensional vector of `64` elements.
//    ::Single(A)       : one-dimensional.
//    ::Double(A, B)    : two-dimensional.
//    ::Triple(A, B, C) : three-dimensional.
let inputs = tensor::Shape::Single(64);

// 3. Define the network.
//    This needs to be mutable (`mut`).
let mut network = network::Network::new(inputs);
```

Importantly, creating the network with a specific input shape allows the framework to infer shapes when adding layers automatically.

Layer definition

Once the network has been defined (Listing 3.7), we can define its layers; a few example layers are presented in Listing 3.8. Here, we assume a `network` to be predefined (and consequently the `tensor` module to be imported).

Listing 3.8: Layer addition. All layers automatically infer relevant shapes. The max-pooling and deconvolutional layers and, of course, the feedback block are omitted here.

```
// 1. Import the `Activation` structure.
use neurons::activation::Activation;

// 2. Add a dense layer.
```

```
network.dense(
    100,       // Number of nodes.
    Activation::ReLU, // Activation function.
    false,     // Should a bias term be added?
    Some(0.1) // Dropout percentage (when training)?
);

// 3. Add a convolutional layer.
network.convolution(
    5,         // Filters.
    (3, 3),    // Kernel.
    (1, 1),    // Stride.
    (0, 0),    // Padding.
    (1, 1),    // Dilation.
    Activation::ReLU, // Activation function.
    None,      // Dropout percentage (when training)?
);
```

Worth noting is the procedural layer definition. That is, the order of layer insertions directly reflects the information flow. Consequently, assuming the network to be predefined as in Listing 3.7, we obtain that information will firstly be reshaped from $\mathbb{R}^{100}$ to become three-dimensional (as described in Subsection 3.2.1): $\mathbb{R}^{1\times10\times10}$, before convolved to produce a final output of $\mathbb{R}^{5\times8\times8}$.

Feedback definition of COMBINE

Assuming a predefined network, we can easily incorporate feedback connections of COMBINE, given the constraint of matching shapes (as defined in Section 3.1). An example connection between the output of the second layer and the network's input is seen in Listing 3.9. In addition to expressing the layer indices of the connection, we state the recursion depth (in this case 1), gradient scale as a function of the recursion depth, and whether to include input-to-input connections (as in Figure 3.3).

Listing 3.9: Example definition of a feedback connection implementing COMBINE.

```
// Import the `Arc` structure.
use std::sync::Arc;

// Feedback connection implementing COMBINE.
network.loopback(
    2,         // From layer X's output.
    0,         // To layer Y's input.
    1,         // Number of loops.
    Arc::new(|_loops| 1.0), // Gradient scaling.
    false,     // Input-to-input skip-connections?
);
```

Importantly, we omit any configuration regarding output-to-output connections

(as in Figure 3.4), as these are automatically included as a consequence of *value* combination, as explained in Subsection 3.2.4.

While the network is predefined with default accumulation methods, we may override this, as seen in Listing 3.10. Note that the accumulation methods are network-wide, *i.e.*, equal for all COMBINE and skip connections, respectively.

Listing 3.10: Accumulation method of skip connections and *value* combination.

```
// Import the `feedback` module.
use neurons::feedback;

network.set_accumulation(
    feedback::Accumulation::Add,  // Skip accumulation.
    feedback::Accumulation::Mean, // Value accumulation.
);
```

Feedback definition of COUPLE

On the other hand, we define feedback connections implementing COUPLE as an individual layer of the network (as described in Subsection 3.2.4). A simple feedback block of three layers (before unwrapping) is seen in Listing 3.11.

Listing 3.11: Example definition of a feedback block implementing COUPLE.

```
// Import the `feedback` module.
use neurons::feedback;

network.dense(256, Activation::ReLU, false, None);

// Feedback connection implementing COUPLE.
network.feedback(
    vec![
        feedback::Layer::Dense(
        //  n    act. function     bias?  dropout
            124, Activation::ReLU, false, None
        ),
        feedback::Layer::Dense(
            86, Activation::ReLU, false, None
        ),
        feedback::Layer::Dense(
            256, Activation::ReLU, false, None
        ),
    ],          // Layers of the feedback block (compacted)
        .
    2,          // Number of loops.
    true,       // Input-to-input skip-connections?
    false,      // Output-to-output skip-connections?
    feedback::Accumulation::Mean, // Couple method.
);
```

To comply with shape constraints, we ensure that the input to the feedback block matches its output. In Listing 3.11, we guarantee this by stating the penultimate layer (`network.dense(...)`) to return an equally shaped output as the subsequent feedback block; in the example, this shape is $\mathbb{R}^{256}$.

3.3 Tools used

Ironically, as this thesis studies and documents AI, certain SOTA models have been of help with both implementation and writing. The following paragraphs will detail their integration into work, with accompanying prompt examples (where relevant).

Before progressing, however, we should give credit where credit is due. Namely, as the accompanying framework is vast in size and functionality, it would have been magnitudes more time-consuming to create without the help of AI (as a solo project, given the time constraints related to the thesis). Although the project started some weeks before the general thesis work, the tools detailed subsequently significantly boosted general efficiency.

Copilot

While the use of AI tools is under scrutiny, and its effect presumably not yet fully understood concerning learning outcomes, we chose to take advantage of the available tools. Integrating such tools might be more harmful, especially relevant for persons new to the field and programming in general. Still, with a thorough programming knowledge, we instead passively rely on AI to boost efficiency, simultaneously being critical and thoroughly understanding the code written.

With this in mind, GitHub Copilot [71], an autocompletion assistant, was activated throughout the implementation of the accompanying framework. Although suggestions are presented regularly, we only sparsely accept them and, in most cases, modify and correct accepted suggestions. This sparsity follows that the codebase is too big for GitHub Copilot to keep in context, therefore not always yielding correct suggestions. As mentioned, being extremely critical of suggestions was, thus, a necessity.

For simple tasks, however, GitHub Copilot [71] was of great help. As no direct prompting is necessary to utilise the tool, exact examples are difficult to produce. But, imagining a for-loop is to be created, one would begin typing as in Listing 3.12:

Listing 3.12: Initial Copilot prompt.

```
1 for i in
```

and wait a few milliseconds for a suggestion to pop up. The suggestion would then reflect the context around the relevant line, in some cases completing the entire content of the for-block, and in others simply the line in question. Imagining a

context of a variable holding an array, `array = [ 1, 2, 3 ]`, the autocomplete suggestion could for instance be that of Listing 3.13:

Listing 3.13: Generated Copilot response.

```
for i in array.iter() {
    println!("{i}");
}
```

leading to a significant amount of time saved throughout implementation. In simple terms, GitHub Copilot was most helpful in suggesting boilerplate code, consequently leading to relatively more time and effort spent by us (the user) on more challenging parts where GitHub Copilot is not as proficient or helpful.

Chatbots

In addition to the integrated coding assistant, chatbots such as ChatGPT [72, 73] and Claude [74, 75] were used. In contrast to the autocompletion models, these reply to explicit prompts. Consequently, we have more room to guide the nature of responses.

To maintain integrity, we rely on chatbots to make suggestions and improvements rather than generating new content from scratch. In other words, we rely on these tools to point out mistakes in either code snippets or paragraphs of text and make suggestions if relevant.

Following this, prompting would follow the format of

> Given the implementation of { *code snippet* },
> point out weak parts along with your reasoning.
> In addition, suggest changes to improve and fix errors.

for coding, and

> Given the text: { *text snippet* }.
> Comment on weak wording or sentences.
> Make suggestions to improve coherence and an academic style.

or

> Your goal is to improve readability and coherence.
> You should ignore commenting on latex-specific notation.
> You should not use overly excessive complex language.

for writing.

Additionally, when it comes to code, chatbots were a great source of error handling. Namely, in contrast to navigating the web, prompting for possible solutions to errors caused by code proved to be highly efficient. While Rust provides excellent error messages, they are not always correct. In these cases, ChatGPT [72, 73] and Claude [74, 75] were of great help, as they could be prompted with the code in question along with the error message, and compactly describe the cause of the error and how to handle it. Being new to Rust, this was an excellent source for broadening my general understanding of challenging aspects of the language, as the chatbot was a great tutor. With this in mind, an example prompt could resemble:

> Running the Rust code: { *code snippet* },
> I get the error { *error message* }.
> Explain why this error occurs and propose a solution.

Grammarly

During writing, the spell-checker Grammarly [77] was used, utilising its free tier. Consequently, the available tools through Grammarly were mainly auto-corrections.

Since Grammarly [77] was accessed through its free tier, the chatbots mentioned above were prompted to serve as a comparable no-cost alternative to Grammarly's paid features.

4 Results

In this chapter, we evaluate feedback connections introduced in Chapter 3. Our investigation highlights how the choice of feedback mechanisms impacts learning by quantifying performance across differing tasks. We include the standard FFNs without feedback connections as a comparable baseline.

Our results demonstrate that employing feedback connections can, in some instances, boost model performance. We find that the chosen feedback mechanisms (*i.e.*, COMBINE or COUPLE, with varying numbers of iterations) differ in performance across tasks. Furthermore, we quantify the importance of feedback connections in the trained networks by probing these and consequently assess whether the networks successfully (learn and) utilise said connections.

4.1 Datasets

Before defining the specific architectures used and the results achieved, we briefly describe various datasets the models were trained on. While previous studies of feedback connections by Liao and Poggio [5] and Caswell, Shen, and Wang [6] mainly focus on multi-dimensional inputs (*i.e.*, images), we here investigate a broader set of problem settings. Our experimental validation employs increasing complexity data for both classification and regression. This allows us to examine the effectiveness of feedback connections across different settings and data structures.

Iris

A typical dataset for simple classification models is the iris set [78]. This dataset consists of 150 records of four features each (sepal and petal measurements), where each record is assigned to one of three classes (iris species). Without going too in-depth, it is worth noting that the three different classes are not fully linearly separable.

While the dataset is a great starting point, it is generally unsuitable for deep networks. This follows from both its limited size and few features. However, the iris set is an excellent validation of our algorithms and the `neurons` framework [4].

As mentioned, a single sample (row) of the dataset has four attributes, which map to one of three classes. Therefore, our architecture must reflect this by having four

input nodes[1]. The hidden layers, however, can be freely chosen to hold an arbitrary number of nodes. As for the inputs, the output layer must represent the three possible classes. Here, we have somewhat more freedom, as either one or three output nodes could effectively represent three classes, depending on the chosen activation function $\sigma_{\text{out}}(\cdot)$. We choose three output nodes, consequently allowing us to use the softmax activation function.

Bike-sharing

Comparable to the iris dataset, another popular yet relatively small dataset commonly used for validation is the bike-sharing set [79]. This dataset contains an increased number of samples opposed to both iris and fourier-transform infrared spectroscopy (FTIR), at 17 389 records. Each sample contains 13 features consisting of measurement time, corresponding weather data, and a single target variable representing the number of rented bikes per hour used for regression.

We hereafter refer to this dataset as both *bike-sharing* and *bike*, interchangeably.

Fourier-transform infrared spectroscopy

While the iris and bike-sharing datasets are great for validation, their small feature space makes them somewhat unconventional for DL. With this in mind, we include a dataset of FTIR measurements. The dataset, which originates from Kristoffersen et al. [80] and is used by Helin et al. [81], is preprocessed through Savitzky-Golay (SG) and extended multiplicative signal correction (EMSC), compactly notated as SG-EMSC.

The FTIR dataset of Helin et al. [81] contains 529, 174 and 182 records of, respectively, training, test and validation samples. Each record has 571 features and is classified into one of 28 possible classes (by-product/enzyme combinations). In addition to the class label, the FTIR dataset contains a scalar target (continuous average molecular weight) used for regression.

With the tabular dimensions, as mentioned, we have similar architectural constraints regarding the number of input and output nodes as with the iris and bike-sharing sets. However, it is unsuitable to define the output layer as a single node, as we may for the iris set, due to the increased number of possible classes. Following this, the arguably best choice for the output layer is 28 nodes along with the softmax activation, resulting in a probability distribution across the possible classes. For regression, as for the bike-sharing set, a single linearly activated output suffices.

Additionally, while validating networks implementing our feedback connections on the FTIR dataset is of interest in itself, we can further extend our analysis by comparing our performance against the results presented by Helin et al. [81].

[1] Of course, given that we do not feature engineer additional (composite) attributes before training the networks.

This comparison allows us to assess the effectiveness of our feedback connections concerning SOTA solutions. By doing so, we additionally validate our claims of a stable framework.

4.2 Architectures

To compare equivalent networks with and without feedback connections, we are constrained by the layer shapes, as presented in Chapter 3. Consequently, we define architectures with several identical layers sequentially, allowing us to easily incorporate skip and feedback connections into the corresponding base architecture. An overview of the base architectures used for the different tasks can be seen in Table 4.1.

DATASET	**BASE MODEL** with named layers				
	input	**1**	**2**	**3**	**output**
Iris	`[4]`	→ `[25]`	→ `[25]`	→ `[25]`	→ `[3]`
Bike-sharing	`[12]`	→ `[24]`	→ `[24]`	→ `[24]`	→ `[1]`
FTIR dense	`[571]`	→ `[128]`	→ `[256]`	→ `[128]`	→ `[1 | 28]`
FTIR convolutional	`1x571`	`1x(9×1)`	→ `1x(9×1)`	→ `[32]`	→ `[1 | 28]`

Table 4.1: Base architectures for the different datasets. Where `[A]` represents a dense layer of `A` nodes, and `Bx(C×D)` represents a convolutional layer of `B` kernels of size `C×D`. For the FTIR models, the output layer is either a single node or 28, respectively, for regression and classification.

The feedback connections are defined from the output of the penultimate layer into the second layer (*i.e.*, `3` to `2`) for all models except the convolutional FTIR, which has feedback connections from the second layer into the first (*i.e.*, `2` to `1`).

Architectures implementing feedback connections of COMBINE follow the structure of Figure 3.4, as a consequence of *value* combination, mentioned in Section 3.2.4. Models implementing COUPLE follow the structure of Figure 3.2. While the framework allows input accumulation (as in Figure 3.3) for feedback connections, our results are generated without this addition.

Every model is defined twice: once with skip connections bypassing the feedback loops, as in Figure 3.5, and once without.

For the FTIR task, we use the same dense architecture used by Helin et al. [81], allowing for one-to-one comparisons. However, the convolutional architecture for the same dataset is modified slightly to allow feedback connections. Namely, instead of a single convolutional layer of eight filters (as is used by Helin et al. [81]), we use two consecutive convolutional layers of one filter each, as presented in Table 4.1. By implementing these modifications, we ensure that the input to the network is of equal shape as the input to the hidden dense layer, thus allowing feedback connections. Additionally, the dense networks of Helin et al. [81] implement batch normalisation.

Our framework does not currently support this, so it is omitted from our models. We use the same activation functions and optimisation technique.

Loss and optimisation

All classification tasks utilise softmax output activation with cross-entropy loss, whereas all regression tasks rely on RMSE loss with a single linearly activated output. All models were optimised through Adam, with respective hyperparameters presented in Table 4.2.

DATASET	LEARNING RATE	BETA 1	BETA 2	EPSILON
Iris	0.0001	0.95	0.999	1×10^{-7}
Bike-sharing	0.01	0.9	0.999	1×10^{-4}
FTIR	0.001	0.9	0.999	1×10^{-8}

Table 4.2: Adam optimiser hyperparameters for the different datasets.

Hardware

All experiments were conducted on an Apple MacBook Air with an M2 chip and 8 GB RAM.

4.3 Training dynamics

This section presents the performance of our networks on the aforementioned datasets (Section 4.1). All models are trained on the respective datasets five times. The results are averaged across these individual runs, providing a representative training dynamic. In addition, we include the confidence interval of $\langle 20\%, 80\% \rangle$ as a shaded region, signifying the lower and upper percentiles.

Figure legend descriptions can be seen in Table 4.3. As mentioned, all models are defined twice: once without skip connections and once with a bypassing skip connection. The figures are consistently organised in a two-column layout, where the left column displays models without bypassing skip connections, and the right column shows the corresponding models with bypassing skip connections. The x-axes per column are shared, and the y-axes are shared across columns.

4.3.1 Iris

Figure 4.1 presents the validation dynamics of models during training on the iris dataset. As mentioned, these results serve primarily as validation of the accompanying framework. Consequently, we observe pretty large and overlapping confidence

LEGEND	DESCRIPTION
FFN	Base model; FFN architecture, without feedback connections.
COMBINE	Base model with feedback connection of COMBINE.
COUPLE	Base model with feedback connection of COUPLE.
x{2..4}	Unwrap depth. *E.g.*, x3: three iterations through the feedback loop.

Table 4.3: Legend descriptions for metric figures.

intervals, suggesting relatively equal performance across models; to some extent, all models seem able to learn and converge towards the same level.

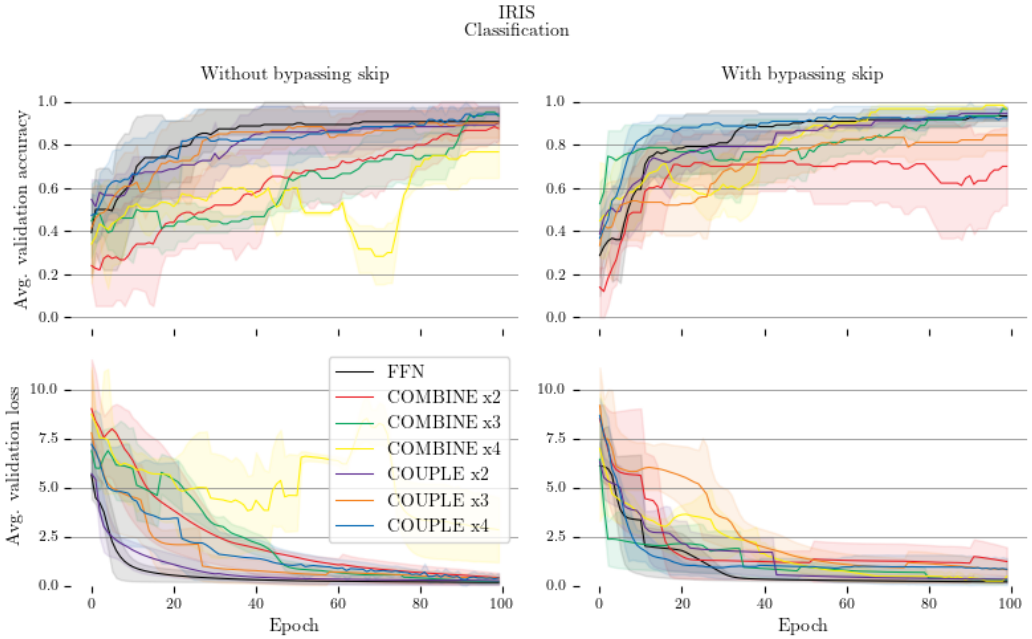


Figure 4.1: Iris classification validation metrics from training. Cross-entropy loss.

In general, we see that models implementing COUPLE are comparable to the base FFN, whereas those of COMBINE take longer to converge with somewhat more unstable patterns. However, due to the considerable confidence interval overlap, most models are relatively similar in performance, likely due to the limited size of the dataset.

4.3.2 Bike-sharing

In the validation metrics for the bike-sharing dataset, shown in Figure 4.2, we observe that feedback connections using COMBINE are entirely unable to learn. This may be due to a poor architectural choice or difficulty escaping a local minimum of the loss landscape. The COUPLE models, on the other hand, converge somewhat faster than the base model but reach a similar performance range.

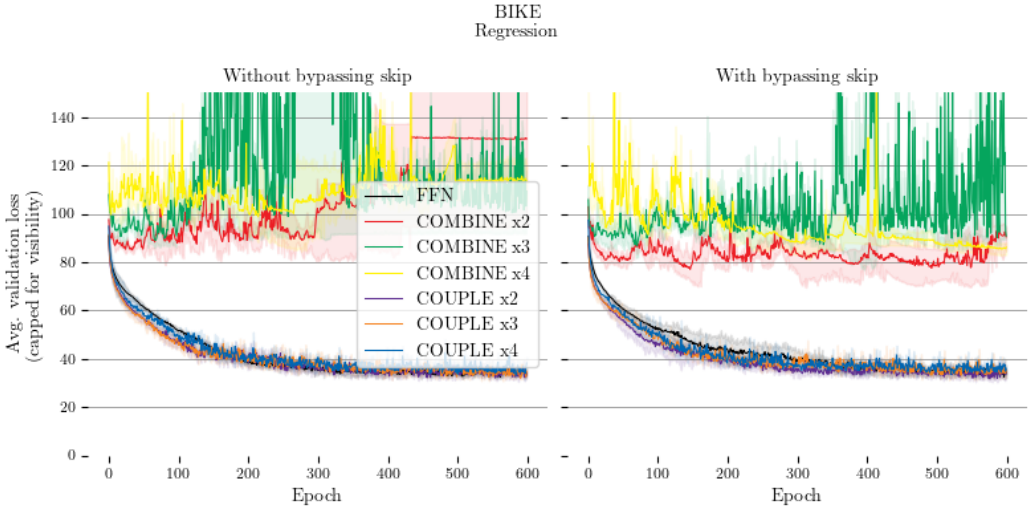


Figure 4.2: Bike-sharing regression validation metrics from training. RMSE loss.

Moreover, all feedback models display more significant loss fluctuations than the more stable base model. Disregarding the unsuccessful COMBINE models, we see that the difference of fluctuations in COUPLE versus the base model, while different, is relatively small.

4.3.3 Fourier-transform infrared spectroscopy

Dense

The base dense FTIR model performs significantly worse than all feedback models for regression (Figure 4.3a). Furthermore, increasing the iteration count of feedback models further speeds up the (number of epochs necessary for) convergence. Nevertheless, after enough epochs, all models converge towards the same range.

In contrast, the classification task (Figure 4.3b) reveals that the base models outperform all those with feedback connections. Here, increasing the iteration count of feedback models reduces the converged performance. *I.e.*, both COMBINE and COUPLE models with two iterations significantly outperform their three-iteration variants. The inclusion of bypassing skip connections, however, narrows this gap somewhat. Additionally, we observe that COMBINE models show great instability throughout their training, along with slower convergence, as opposed to both the base and COUPLE models.

Convolutional

Considering the FTIR convolutional regression models (Figure 4.4a), COMBINE models generally display large confidence intervals, with the COMBINE model of three iterations showing the most significant variance and instability. However, the same model of two iterations is entirely stable without a bypassing skip connection

and relatively stable with this included, compared to its three-iteration variant. All COUPLE models outperform the base model with confident predictions.

Unlike the dense regression models (Figure 4.3a), the COUPLE convolutional models converge to a lower loss than the base model. This is also true for the COMBINE model with two iterations without the bypassing skip connection.

Inspecting the FTIR convolutional classification models (Figure 4.4b), focusing on those without bypassing skips, we observe that all models except the COMBINE model with two iterations converge to the same performance level. However, feedback models of COUPLE converge progressively earlier with increasing iteration counts. In contrast, the COMBINE model converges more slowly than the base model.

The addition of skip connections that bypass layers negatively affects classification model performance. Though the COUPLE models maintain their performance compared to the base model, the performance difference between models using two versus three iterations becomes smaller than the case without skip connections. All models with bypassing skip connections show more unstable behaviour patterns than those without.

4.4 Probing the trained networks

To gain deeper insights into the internal dynamics of our trained models, we systematically investigate the information flow through the possible pathways. Our analysis, focusing on severing existing skip- or feedback connections, allows us to understand how these contribute to the performance of our models. Similarly to the training metrics, the probed distinction between approaches follows the format of Table 4.3.

For investigating the effects of skip connections bypassing feedback loops, we clear the `connect`-attribute, thus severing these pathways for subsequent evaluation. Similarly, the impact of feedback connections can be analysed by effectively transforming the network into a shallow (compact) version. Consequently, evaluating the shallow representations directly reflects the importance of iterative processing in relation to performance.

Our analysis involves validating the modified networks on the same validation set as the original networks were tested on. We can thereafter compare the performance impact of these connections. All results are presented as $\mu \pm \sigma$, where μ is the mean and σ is the standard deviation, calculated across the five individual runs. Table 4.4 presents an excerpt of the probing applied to the FTIR dense models for classification. A complete overview of the probed results can be found in Appendix B.

In general, severing the bypassing skip connection does not significantly alter performance, as seen in Table 4.4. Surprisingly, some cases of severing the bypassing skip connections even improve performance. In the excerpt of Table 4.4, all models trained with bypassing skip, except COUPLE with three iterations, benefit from its severing (post-training).

MODEL	ORIGINAL Accuracy Loss	SKIP OFF Accuracy Loss	FEEDBACK OFF Accuracy Loss
FFN w. bypassing skip	0.9846 ± 0.0041 0.0788 ± 0.0131	0.9977 ± 0.0028 0.0028 ± 0.0025	
COMBINE x2 w. bypassing skip	0.9626 ± 0.0101 0.2520 ± 0.0877	0.9793 ± 0.0148 0.0725 ± 0.0547	0.1425 ± 0.0935 5.2835 ± 0.7749
COMBINE x3 w. bypassing skip	0.9352 ± 0.0204 0.8481 ± 0.2875	0.9471 ± 0.0067 0.6425 ± 0.0996	0.3046 ± 0.0717 3.4350 ± 0.5968
COUPLE x2 w. bypassing skip	0.9714 ± 0.0081 0.2423 ± 0.0552	0.9897 ± 0.0067 0.0423 ± 0.0259	0.0276 ± 0.0245 3.6988 ± 0.1918
COUPLE x3 w. bypassing skip	0.8242 ± 0.0628 2.1106 ± 0.7940	0.8034 ± 0.0794 2.2961 ± 1.0132	0.0230 ± 0.0163 3.5176 ± 0.2149

Table 4.4: Excerpt from probed results of FTIR dense for classification. Cross-entropy loss. The mean and standard deviation are calculated over five runs.

On the other hand, omitting the feedback connections dramatically reduces performance. In some cases (*e.g.*, feedback models of COUPLE), leading to worse results than random guessing. Consequently, we obtain that all models heavily rely on the iterative behaviour of feedback connections; the models, in other words, correctly learn to iterate on their latent representations of inputs before providing their final outputs.

4.5 Time usage

Table 4.5 presents an excerpt of the (wall clock) time differences of running the various FTIR convolutional models for classification. Appendix C provides a complete overview of time comparisons.

MODEL	TRAIN	VALIDATE
FFN	29.0975 ± 2.2787	7.0207 ± 0.3973
COMBINE x2	32.6733 ± 0.4177	11.9375 ± 0.0946
COMBINE x3	39.4026 ± 4.3102	17.2540 ± 0.4477
COUPLE x2	47.6980 ± 0.8026	13.2830 ± 0.9639
COUPLE x3	51.0296 ± 0.5958	16.7820 ± 0.3534

Table 4.5: Excerpt from time differences of FTIR convolutional models for classification. All times are in milliseconds. The mean and standard deviation are calculated over five runs. The training times are obtained from one epoch.

As expected, we see a significant jump of 45.98% and 29.51%, comparing models

with two and three iterations, respectively, in (per-epoch) training time between models of COMBINE and COUPLE, the latter being the more computationally intensive.

Contrastingly, the validation times of COMBINE versus COUPLE are almost identical. Their inequality is a direct consequence of combining *values* in COMBINE (which is done through the forward pass[2]) and COUPLE, which has to keep unwrapped *values* in memory.

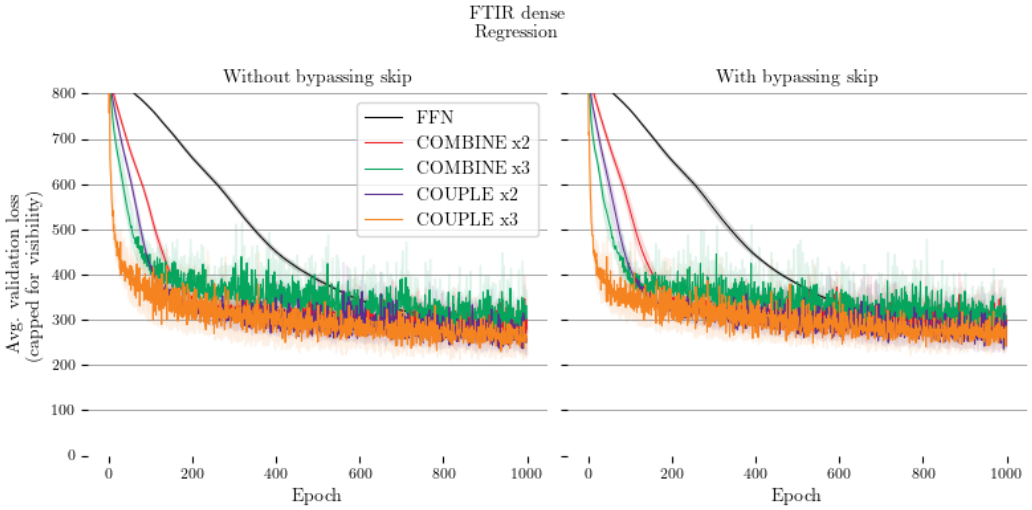
Notably, the time usage of feedback models is directly related to the unwrapped size. *I.e.*, feedback loops spanning many layers or with excessive iteration count result in a more significant increase in spent time. Consequently, we chose relatively short feedback connections to prevent extensive training times. Furthermore, we highlight that the layer types in the feedback loop directly influence time spent, with computations a function of the data shapes and, consequently, the number of operations.

Time-weighted training dynamics

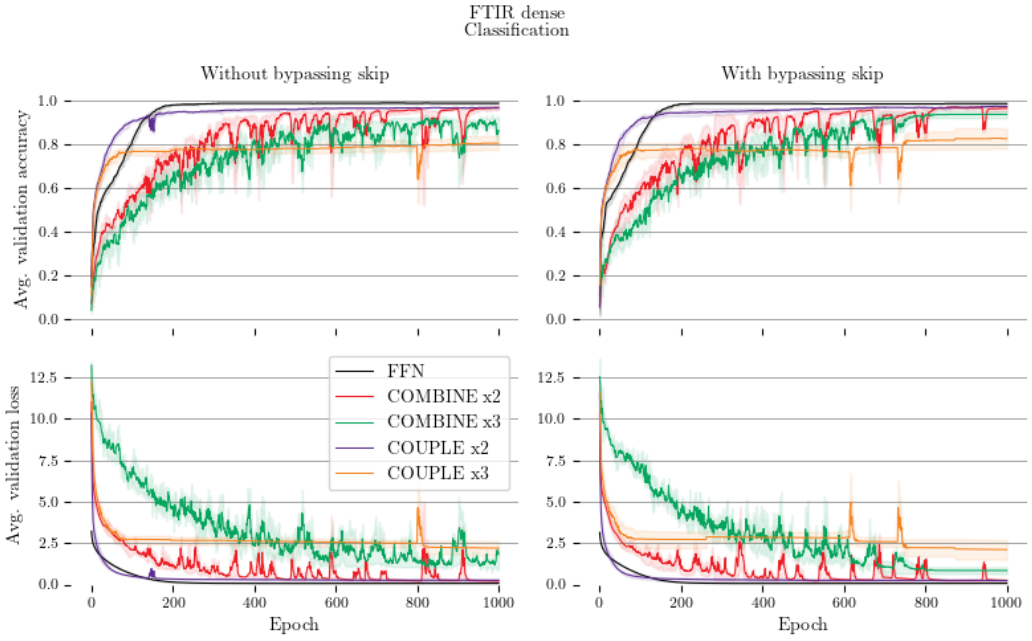
Using the average per-epoch training times for each model, we calculate the weighted validation curves, as in Figure 4.5 (a complete overview is found in Appendix D). This provides a more realistic comparison between models, as we then account for computational cost. Comparing the weighted dynamics of Figure 4.5 with the corresponding unweighted dynamics of Figure 4.2, we see that the base model outperforms all others when time is accounted for.

While this analysis generally reduces the performance gap between base and feedback models, feedback models maintain their advantage in several cases.

[2]This may easily be turned off with a boolean flag representing whether the network is training.

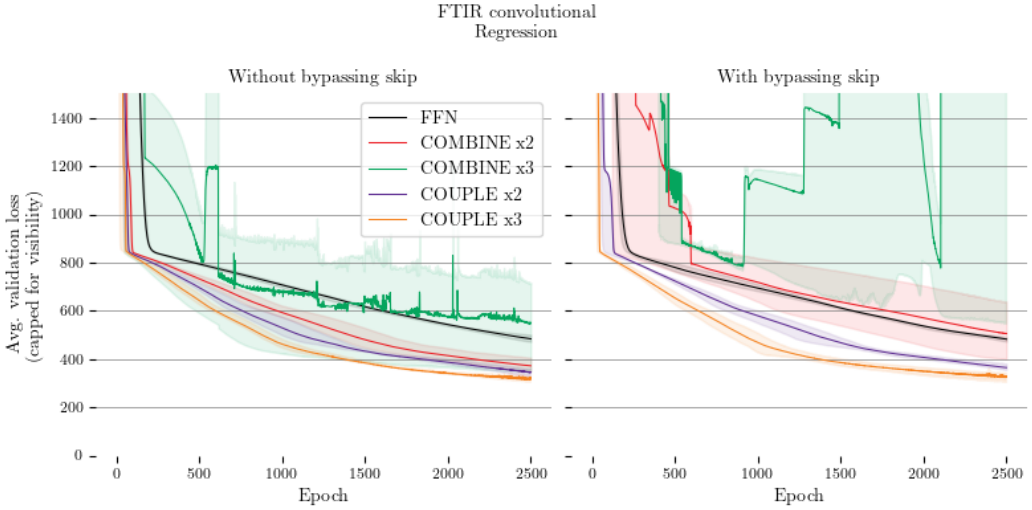


(a) Dense FTIR regression validation metrics from training. RMSE loss.

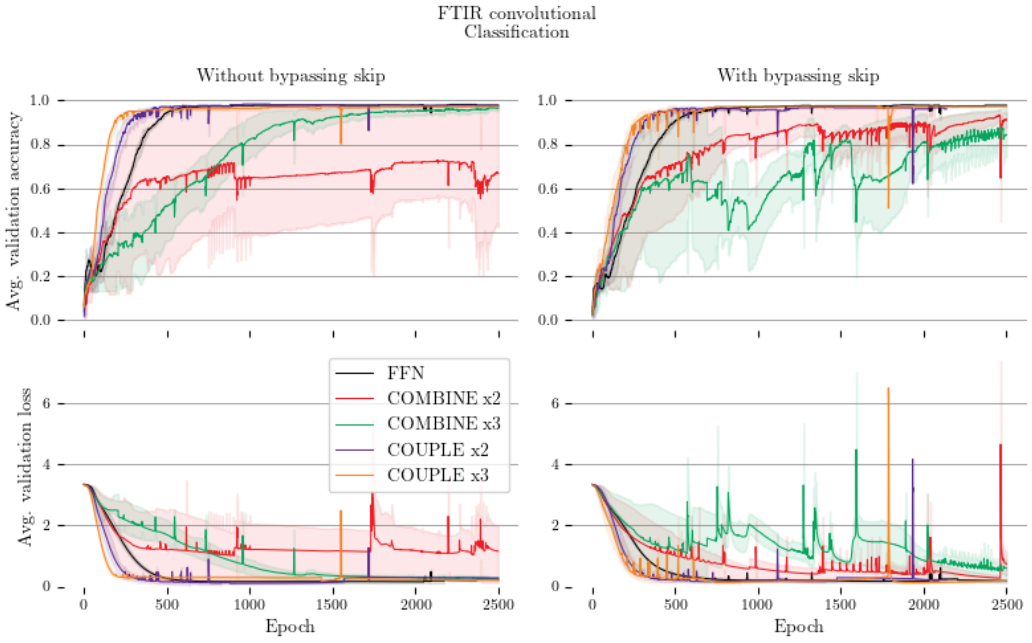


(b) Dense FTIR classification validation metrics from training. Cross-entropy loss.

Figure 4.3: Dense FTIR validation metrics.



(a) Convolutional FTIR regression validation metrics from training. RMSE loss.



(b) Convolutional FTIR classification validation metrics from training. Cross-entropy loss.

Figure 4.4: Convolutional FTIR validation metrics.

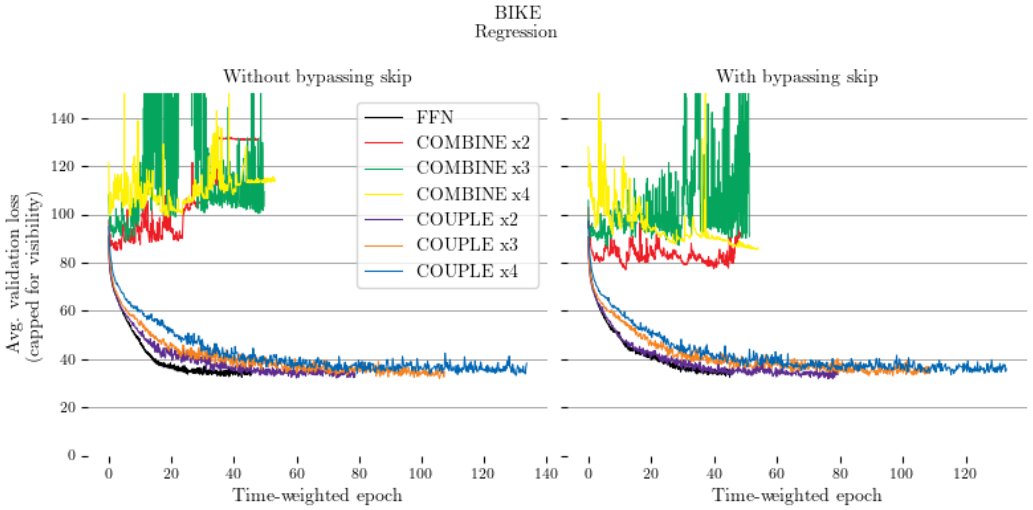


Figure 4.5: Time-weighted bike-sharing regression validation metrics from training. RMSE loss.

5 Discussion

Although we use small validation datasets compared to SOTA studies by Liao and Poggio [5] and Caswell, Shen, and Wang [6], our results establish a promising foundation for more comprehensive future studies and demonstrate proof-of-concept. That is, we prioritised the development of a thorough from-scratch framework, and therein were full possibilities of tweaking the network structures over corporate-grade scaling and deployability. The custom-built `neurons` framework by Lavik [4], in other words, successfully demonstrates the fundamental principles of feedback connections, though certain performance constraints are observed.

The benefits of feedback connections vary across tasks, showing minimal improvements for simple datasets like iris [78] but significant performance gains for more complex datasets such as FTIR [81]. Although one might obtain similar improvements with increasing the learning rate in conventional FFNs, our empirical analysis reveals that removing feedback connections consistently leads to performance degradation, thus validating their essential role in the network architecture.

Notably, the recursion depth (number of iterations through the feedback loop) does not inherently correlate to performance. This becomes evident through the contrasting results of the FTIR task: the dense model of COMBINE is significantly better performant with two loops compared to three loops (Figure 4.3b), the convolutional model oppositely has improved performance with the additional loop iteration (Figure 4.4b).

Our findings indicate that feedback connections improve convergence time across most experimental conditions, except for the iris dataset, without bypassing skip connections (Figure 4.1). Predominantly through the FTIR [81] task, our feedback models outperform the comparative base model. In particular, the regression models of the same dataset display significant improvements (Figures 4.3a and 4.4a).

However, accounting for per-epoch time spent during training, this gap between convergence times is altered. Consequently, models trained on smaller datasets suggest that including feedback connections does not lead to quicker convergence. For more complex datasets (*i.e.*, FTIR), however, the gap remains in favour of feedback models (especially for the FTIR regression task).

5.1 Omitted bypassing skip connection

As mentioned, skip connections are uncommonly used in small networks. However, by including these, we introduce alternative paths for information flow through said networks. As this thesis aims to investigate feedback connections for layer reuse, probing the bypassing skip connections post-training allows us to assess whether these feedback connections are utilised.

In the following analysis, we purely focus on feedback models; we disregard the results related to the base models, as these do not implement layer reuse.

Impact on accuracy

Removing bypassing skip connections from trained models generally resulted in minimal performance variations across most experimental configurations. The notable exception was observed in the COMBINE iris network with two iterations, where the removal led to a substantial 39% reduction in accuracy, contrasting sharply with the typically lower accuracy fluctuations observed in other models when skip connections are removed.

Interestingly, eliminating skip connections yields unexpected performance improvements for many models. This behaviour is somewhat counterintuitive; we expect models trained with bypassing skip connections to utilise these. Furthermore, their respective standard deviations remain small, suggesting statistical significance rather than experimental artefacts. The observed performance improvements, though small, reveal important insights about the network's internal dynamics; these findings indicate that the models rely heavily on their latent representations rather than the input context. This behaviour is particularly noteworthy in feedback-based architectures, where the recursive iteration of latent representations leads to substantial input processing, effectively reducing noise in the original signal, which is reintroduced through the skip connection. For these cases, the additional skip connection during training can thus be thought of as a regularisation technique.

The minimal performance difference after removing the bypassing skip connections suggests these networks learn to effectively ignore these connections during training. This indicates that the models primarily rely on their learned latent representations rather than the raw input data.

Impact on loss

Models where removing bypassing skip connections improved classification performance also show significantly reduced loss values. Even though loss and accuracy correlate, the substantial decrease in loss indicates enhanced confidence in class predictions. This effect was particularly evident in the FTIR classification models (Table 4.4), where the loss was reduced with 83% in the dense COUPLE model with

two iterations when the bypassing skip is removed post-training.

The dramatic reduction in loss suggests that models make more confident predictions when relying exclusively on their processed representations rather than incorporating raw input data through skip connections. This observation supports our hypothesis that skip connections, in these cases, function as a regularisation mechanism. Furthermore, it indicates that models trained without the bypassing skip connection likely become trapped in some local minima within the gradient landscape.

Unlike classification tasks, where removing skip connections sometimes improves accuracy, regression tasks show no such improvements. Instead, most regression models show slightly worse performance without skip connections (Appendix B). This suggests that regression models depend more on the raw input data than classification models. However, it is essential to note that these performance drops are much smaller than the significant drops we see when removing feedback connections.

5.2 Omitted feedback connection

Unlike omitting skip connections, omitting the feedback connections leads to a severe drop in performance. The magnitude of this performance drop directly correlates with the number of iterations (recursion depth) (especially for COUPLE models and, to some extent, COMBINE models); models trained with more feedback iterations generally experience more considerable performance losses when these connections are removed. This relationship is expected, as these models were specifically designed to enhance their results through iterative processing of internal representations. Consequently, the models perform significantly worse when we omit their ability to refine predictions through multiple iterations (post-training).

The impact of removing feedback connections varies across tasks, but models implementing COUPLE show an extreme dependence on this iterative mechanism. Most of these models with removed feedback connections result in performance converging towards randomised predictions for increased recursion depth.

While still showing substantial performance reduction, COMBINE models maintain relatively higher performance than COUPLE models when feedback connections are removed. This suggests COUPLE more effectively leverages feedback mechanisms during regular operation. COMBINE models exhibit more variable performance across different iteration counts than COUPLE models. As shown in Table 4.4 (and in Appendix B), deeper COMBINE models sometimes outperform their shallower counterparts when feedback connections are removed. This contradicts the intuitive expectation that models trained with more iterations should show proportionally larger performance drops when probed post-training, which is the case for COUPLE.

5.3 Time-weighted convergence

COMBINE models demonstrate comparable wall clock time usage to base models, with only minor variations due to additional forward pass computations (as described in Chapter 3). Furthermore, training (and validating) COMBINE models of various iteration counts also remain relatively stable as opposed to their more computationally demanding COUPLE alternatives.

Interestingly, per-epoch training times of COMBINE models sometimes outperform the base FFNs. However, as the datasets used are pretty small and consequently lead to relatively few operations (as opposed to larger datasets such as cifar-10 [82]), these results presumably do not remain stable when upscaling. That is, as feedback models result in an increased number of operations, increasing the size of either networks or datasets must intuitively lead to more time spent, which to some extent seems to be negligible in our small examples.

COUPLE implementations show substantial increases in computational overhead, with training times increasing up to 281.8% compared to base models (without bypassing skip connections, Table C.1). This significant difference is a direct consequence of backpropagating through the unwrapped representation, and we observe an equivalent increase in time spent per-epoch training concerning its depth. However, during validation, the performance gap between COUPLE and COMBINE narrows considerably, particularly with double-iteration implementations, where their processing approaches are similar and differ mainly in memory management.

When accounting for training time (Figure 4.5 and Appendix D), we obtain a more realistic view of model performance, which may alter optimal model selection. For example, considering the bike-sharing task (Figures 4.2 and 4.5) where feedback models require fewer epochs to converge when time is unaccounted for, has longer per-epoch training times consequently, the base model achieves faster overall convergence when time is accounted for.

For FTIR regression and convolutional classification, however, COUPLE models maintain their performance advantage over the base model, even when time is accounted for, though with a smaller temporal convergence gap.

5.4 Fourier-transform infrared spectroscopy comparison

In comparison to Helin et al. [81], we reduced the maximum training epochs from 5000 to 1000 for dense models and to 2500 for convolutional models, as our models demonstrated earlier convergence (Figures 4.3 and 4.4). Table 5.1 presents a comparative analysis, where accuracy values represent classification performance and loss values indicate regression RMSE. The overall best model within the two tasks is highlighted with bold font.

	MODEL	ACCURACY	LOSS
Helin et al.	**Dense**	0.948 ± 0.0007	357.0 ± 2.57
Our dense	COMBINE x2	0.9659 ± 0.0153	279.8956 ± 26.8177
	COMBINE x3	0.8604 ± 0.1148	303.6938 ± 27.8487
	COUPLE x2	0.9659 ± 0.0064	261.5397 ± 40.8263
	COUPLE x3	0.8036 ± 0.0377	**250.9078 ± 32.2835**
Helin et al.	**Convolutional**	0.948 ± 0.001	323.0 ± 3.96
Our convolutional	COMBINE x2	0.6662 ± 0.3952	372.0797 ± 28.1221
	COMBINE x3	0.9643 ± 0.0143	551.3221 ± 268.0685
	COUPLE x2	**0.9780 ± 0.0000**	345.7501 ± 15.0615
	COUPLE x3	0.9725 ± 0.0000	323.9352 ± 9.9574

Table 5.1: FTIR metrics from Helin et al. [81, Table 2]. The extracted values are the SG-EMSC rows of the MLP (Dense) and CNN (Convolutional) models. For comparison, we include our feedback models without bypassing skip connections. The overall best results for respectively classification and regression are displayed in bold.

Classification

Comparing our classification results, we find that most feedback models outperform that of Helin et al. [81], with subpar performance only for the dense models implementing COMBINE and COUPLE with three iterations and the convolutional model implementing COMBINE with two iterations.

Our best-performing (classification) model achieves a 3.2 percentage point improvement over the top-performing SG-EMSC model of Helin et al. [81]. While these models are not one-to-one with the article, our (compact) models include fewer trainable parameters (as we have two subsequent one-filter convolutional layers as opposed to their one eight-filter layer, as commented on) along with the mentioned halved number of epochs. However, we observe that the converged performance of feedback models is comparable to our base models. Feedback models implementing COUPLE do, on the other hand, appear to be orders of magnitude more stable, visible through their minor standard deviations.

Regression

All our dense feedback models outperform the convolutional and dense models of Helin et al. [81]. Contrastingly, our convolutional feedback models are all outperformed by Helin et al. [81] on the regression task.

Our regression results show substantial improvements compared to Helin et al. [81]. The dense feedback model of COUPLE with two iterations and a bypassing skip connection achieved the best performance, improving accuracy by 22.3 percentage points. Notably, this was achieved in just 1000 epochs, as opposed to their 5000. Even

compared to their (overall) best (non-SG-EMSC) model, with a RMSE of 316.0 ± 4.34 [81, Table 2], our model demonstrates a 20.1 percentage point improvement. However, the increased standard deviations of our regression model compared to theirs do suggest some instability, also visible in Figure 4.3a; but, since our models were trained for only one-fifth of their number of epochs, we anticipate that these deviations would diminish with extended training time.

Models with bypassing skip

Conversely, we compare our models trained with bypassing skip connections with Helin et al. [81] (our results are seen in Tables B.2 and B.4).

Interestingly, our best-performant classification model is a model that is trained with a bypassing skip that is turned off post-training. Namely, our dense COUPLE model with two iterations: achieving 98.97% (Table B.2), which is a 4.4 percentage point increase over Helin et al. [81, Table 2].

For regression, the same model, with the bypassing skip present post-training, leads to best RMSE: 242.2010 (Table B.4). This is a 25 percentage point decrease over the best SG-EMSC preprocessed data model of Helin et al. [81], and a 23 percentage point decrease compared to the aforementioned best model of Helin et al. [81, Table 2].

Convergence times

The benefits of increased unwrapped depth must be weighed against longer training times. For the FTIR classification task using convolutional architecture (Figure 4.4b and Table 4.5), COUPLE with three iterations increases training time by 75.37% per epoch, resulting in similar overall convergence times compared to the base model (seen in Figure D.4).

On the other hand, the regression task, despite a maximum 67.18% increase in training time (Table C.4, using times without bypassing skip) for feedback models still outperforms the base models even with this accounted for (Figure D.5).

5.5 Limitations and remaining challenges

Output-to-output connections (Figure 3.4) may become ineffective in specific scenarios, particularly when the initial latent representation (B) dominates the feedback signal (B). In these cases, where feedback values approach zero during element-wise summation, feedback loops become redundant and have a negligible impact on the final output (*out*).

However, this theoretical limitation appears minimal in practice, likely due to the inherent variability in data used to train ANNs (presumably especially such as LLMs, which is currently being trained on incomprehensible amounts of data [83]),

suggesting that output-to-output connections remain valuable. Our empirical results (on the relatively tiny datasets) support this, with only one observed instance of potential redundancy (the two-iteration COMBINE model of bike-sharing regression, Table B.6).

Likewise, input-to-input connections (Figure 3.3) may, in some instances, be unwanted. Imagining a network with many feedback iterations, adding the initial inputs to every iteration may consequently regularise the possibility of extreme latent-representation iteration, as the initial input is added at every step, forcing the model to rely heavily on this regardless of the iteration count.

5.5.1 COMBINE

A key limitation of combining *values* before backpropagating is the potential reduction in model explainability and introduced noise into the system. As mentioned, the addition of skip connections within COMBINE models (illustrated in Figures 3.3 and 3.4) presents challenges for backpropagation. Consequently, as we combine *values* during the forward pass, we implicitly include output-to-output connections (as in Figure 3.4). Furthermore, as we backpropagate through the compacted representation, we cannot correctly account for the gradient flow as in Equation 2.18. With this in mind, a possible reason for the, in some instances, slow convergence and inability to learn for COMBINE models could be an effect of this property.

While COMBINE models demonstrate strong performance in specific scenarios, our experimental results (Figures 4.2 and 4.4a) reveal training instabilities and occasional convergence failures, which might be a factor of the aforementioned architectural decision of *value* combination. This behaviour might be explained by examining the fundamental structure of COMBINE. Since backpropagation is applied to combined values, the model may struggle to account for the temporal order of forward propagation properly; the approach lacks temporal context: the network cannot distinguish between initial values and those from subsequent iterations when backpropagating, which may affect its learning capabilities. While COMBINE models can still learn, they typically show two key limitations: slower convergence compared to other models and, in some cases, complete failure to converge. These observations support our understanding that COUPLE is better suited for leveraging feedback connections due to its implicit temporal incorporation.

However, to combat these issues, a potential solution might be to combine *values* in a weighted manner. The simplest case would thus become that of Equation 5.1:

$$v_i = \sum_{j=0}^{m} \frac{1}{m-j} v_i^j \tag{5.1}$$

where temporally earlier *values* are discounted more than later ones.

Overlapping feedback loops

When following Algorithm 3, *value* combination occurs within the main loop, which presents challenges for networks with overlapping feedback connections. Consider the network with four subsystems shown in Figure 5.1: the forward pass first encounters the feedback loop from C to A. According to the algorithm, intermediate *values* of B and C are accumulated before proceeding to D. Subsequently, the connection from D to B triggers another accumulation of the *values* of C.

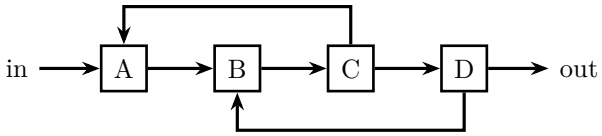


Figure 5.1: A problematic feedback connection for certain accumulation methods used in Algorithm 3.

If *value* accumulation is implemented through averaging, this double encounter creates an uneven weighting, potentially leading to instabilities during backpropagation. With this in mind, we purely consider non-overlapping feedback connections.

5.5.2 COUPLE

While overlapping connections are unproblematic for COUPLE, other downsides present themselves. Namely, backpropagating through the unwrapped representation of the network leads to increasing computational complexity. Consequently, training time is directly linked to the size of the feedback connections, *i.e.*, the unwrapped representation of the networks.

Furthermore, as gradients get diluted through the backward pass, it might be beneficial for stability during training to account for this when coupling weights, as opposed to, for instance, simply averaging them. In this manner, we could instead scale the individual weights by their relative distance to the final output through a weighted sum

$$W_{a,b} = \sum_{i=0}^{m} \frac{1}{m-j} W_{a,b}^{i} \,, \tag{5.2}$$

similarly to the proposed *value* combination of Equation 5.1.

5.5.3 Framework

The current implementation of `neurons` [4] prioritises modularity and proof-of-concept over performance optimisation. While written in Rust for high performance, the framework lacks optimised tensor operations and automatic gradient calculations of established frameworks. Consequently, manual gradient computation and exces-

sive data cloning directly impact computational efficiency, though these limitations can be addressed in future iterations.

5.6 Future work

A consequence of the modularity of `neurons` [4] is the ease of defining variably sized networks. However, practical upscaling[1] requires further development; while the current implementation provides basic functionality, several key aspects require optimisation. The modular structure of `neurons` allows for iterative improvements, which will ultimately enable validation of our feedback theory (Chapter 3) on more extensive networks and datasets.

Furthermore, our results on the FTIR dataset [80, 81] and the corresponding comparisons with Helin et al. [81] suggest that feedback models perform well on regression tasks. While the bike-sharing models produce inconsistent results, we presume the cause is its limited feature space, and the increased complexity of FTIR yields favourable results for feedback models as opposed to those without feedback connections.

Upscaling and potential use cases

Consequently, we predict feedback models to outperform base models on similar datasets to FTIR [80, 81]; that is, datasets of high complexity, where iterative processing can significantly reduce noise in latent representations.

This is the case for the related study by Caswell, Shen, and Wang [6], where their models outperform that of equivalent deep networks on the cifar-10 [82] dataset. We, therefore, expect similar results when upscaling our validation to include the likes of this dataset. Such cases, where it is inherently necessary to heavily process said data to produce accurate outputs, include, but are not limited to, image inputs.

Furthermore, due to the limited size of the datasets used in this thesis, comparisons with equivalent deep networks have been omitted. In future studies of increasing size (*e.g.*, validation on cifar-10 [82]), comparisons of feedback networks with equivalent deep networks become an interesting metric, as presented by Caswell, Shen, and Wang [6], in addition to comparisons with equivalent compact networks.

Stability during training

As we generally observe unstable patterns in COMBINE models, further investigation of said behaviour is necessary. While the cause of this instability might be due to the aggregation function of the *values*, this remains to be seen and should be studied.

[1] We attempted training using cifar-10 [82], highlighting the need for such optimisation as current performance limitations prevent practical implementation at this scale.

Due to the inherent compacted operations of COMBINE during backpropagation, it is particularly advantageous to identify the source of this instability. This is especially true when compute power is limited during training. In such cases, our COMBINE approach offers a way to train effectively deeper networks (*i.e.*, when unwrapped) with significantly reduced computational complexity.

Numerical instabilities

To potentially increase computational speeds, we have chosen to implement the `neurons` framework [4] with 32 bit floating points (`f32`) instead of the larger 64 bit variant (`f64`). Consequently, some instability observed might be an artefact of this choice. For future studies, a simple conversion of all `f32` occurrences with `f64` poses no difficulty and might be beneficial to perform regarding the aforementioned stability issues.

Practical deployment

As `neurons` was developed to display proof-of-concept over practical deployment, the framework omits key parts to achieve the latter. Namely, once networks with COUPLE connections have been trained, these may be compacted to provide a less memory-intensive footprint. Furthermore, trained networks do not need to store their intermediate *values*. While these *values* are negligible for small networks or datasets, removing these is essential for further validation performance improvements (mainly concerning memory and time). Additionally, serialisation (and consequently deserialisation) has not been prioritised and remains unimplemented.

Once the remaining challenges have been addressed, the significant potential for implementing effectively deeper networks in a compact form, particularly for deployment on small devices, becomes apparent. Our feedback models, with their small parameter footprint and simultaneously imitating their deep equivalent, make them particularly valuable for resource-constrained environments, where traditional deep networks might be impractical due to memory and computational limitations.

5.7 Comparison with related work

As mentioned, similar work to ours by Liao and Poggio [5] and Caswell, Shen, and Wang [6] implements feedback connections. However, their approach for backpropagation is more comparable to that of RNNs:

$$G_{a,b} = \sum_{i=0}^{m} G_{a,b}^{i} \,, \tag{5.3}$$

where they aggregate the gradients of the unwrapped network before updating the compacted representation. Instead, we combine the intermediate *values* before backpropagating (COMBINE) or couple the weights after unwrapped backpropagation

and updates (COUPLE). Future studies of our novel feedback methods could thus benefit from a more in-depth comparison with this RNN-inspired approach concerning training speeds and performance validation.

Imagining a network as in Figure 5.2, we can easily compare our approach (namely, COUPLE) to theirs (Equation 5.3).

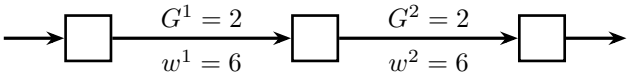


Figure 5.2: Network for comparison with related studies. Abstracted and unwrapped network, where w^1 and w^2 are equal. The two gradients (G^i) correspond to each weight (w^i); their values are arbitrary.

Following the approach of Liao and Poggio [5] and Caswell, Shen, and Wang [6], the updated weight $w = w^1 = w^2$ would become

$$w = w - \eta \times \left(G^1 + G^2\right) = 6 - \eta \times (2 + 2) = 6 - \eta \times 4 \,. \tag{5.4}$$

In contrast, our COUPLE approach (assuming weight coupling is done through averaging) would result in an updated weight of

$$w = \frac{(w^1 - \eta \times G^1) + (w^2 - \eta \times G^2)}{2} = \frac{12 - \eta \times 4}{2} = 6 - \eta \times 2 \,. \tag{5.5}$$

Consequently, we see that our and their approach yields different results. Notably is our similarity to theirs (in the case of our mean aggregation), where doubling our learning rate effectively would lead to equal results. However, our framework is designed to be highly adjustable concerning aggregation functions *et cetera.*, making it easy for future studies to tweak these and consequently alter our similarity with Liao and Poggio [5] and Caswell, Shen, and Wang [6].

Additionally, Caswell, Shen, and Wang [6] includes input-to-input connections (as in Figure 3.3) in their models. As this is easily included in our two approaches by specifying the respective boolean flag, as mentioned, future investigations with this turned on might be beneficial (following their success on the cifar-10 [82] dataset).

6 Conclusion

This thesis presents and validates two novel approaches for incorporating feedback connections into neural networks. These connections enable networks to iteratively refine their latent space representations while reusing existing weights, resulting in a reduced parameter footprint compared to equivalent deep networks without feedback connections.

Our comparative analysis of validation metrics reveals that models with feedback connections generally achieve performance comparable to that of their non-feedback counterparts. A notable finding is the variation in convergence speeds across different architectures. Models implementing feedback connections, particularly those using our COUPLE approach, demonstrate faster convergence than baseline models. For specific problems, feedback models even surpass the performance of baseline architectures. While we generally obtain favourable convergence speeds for our feedback models, accounting for (wall clock) time spent alters this conclusion. Considering the time-weighted convergence, the gap between feedback- and base models is narrowed, in some cases favouring base architectures. However, we generally observe that the gap remains in favour of feedback models for increasingly complex data.

Furthermore, we quantify the significance of the learned feedback loops through systematic probing of the trained models. In all cases, omitting the feedback connection from trained networks leads to a significant loss in performance. The substantial performance degradation observed when removing feedback connections from trained networks confirms that these connections are not only actively learned but also integral to the functioning of these networks.

Moreover, through training feedback models with an additional skip connection bypassing the feedback loop, information may flow through two pathways. Consequently, inspection of these post-training validates our claim that networks significantly benefit from iterative processing, as removing said bypassing connections post-training does not (in most cases) significantly alter performance.

Our implementation utilises the custom-built DL framework `neurons` by Lavik [4], which introduces some computational constraints. While this limits our validation to smaller datasets, the results convincingly demonstrate the potential of feedback connections in improving both convergence speed and model performance. These promising findings establish a foundation for future large-scale studies investigating feedback mechanisms in neural networks. Consequently, future develop-

ment may prove especially beneficial as feedback networks have great potential for resource-constrained environments.

Bibliography

[1] J. Hestness, S. Narang, N. Ardalani, G. F. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou. Deep Learning Scaling is Predictable, Empirically. *CoRR*, 2017. DOI: 10.48550/arXiv.1712.00409. arXiv: 1712.00409. URL: http://arxiv.org/abs/1712.00409.

[2] T. B. Brown et al. Language Models are Few-Shot Learners. *CoRR*, 2020. DOI: 10.48550/arXiv.2005.14165. arXiv: 2005.14165. URL: http://arxiv.org/abs/2005.14165.

[3] J. Hoffmann et al. Training Compute-Optimal Large Language Models. *arXiv*, 2022. DOI: 10.48550/arXiv.2203.15556. URL: http://arxiv.org/abs/2203.15556.

[4] H. H. Lavik. neurons. Version 2.6.2. URL: https://github.com/hallvardnmbu/neurons. 2024.

[5] Q. Liao and T. Poggio. Bridging the Gaps Between Residual Learning, Recurrent Neural Networks and Visual Cortex. *ArXiv*, 2020. arXiv: 1604.03640 [cs.LG]. URL: https://arxiv.org/abs/1604.03640.

[6] I. Caswell, C. Shen, and L. Wang. Loopy Neural Nets: Imitating Feedback Loops in the Human Brain. Tech. rep. Stanford University, 2016. URL: https://www.semanticscholar.org/paper/Loopy-Neural-Nets%3A-Imitating-Feedback-Loops-in-the-Caswell/11603a256a53c0ce80ccf58cf6fc92b5c0e16e07.

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR*, 2017. arXiv: 1712.01815 [cs.AI]. URL: http://arxiv.org/abs/1712.01815.

[8] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-Resolution Image Synthesis with Latent Diffusion Models. *CoRR*, 2022. DOI: 10.48550/arXiv.2112.10752. URL: http://arxiv.org/abs/2112.10752.

[9] Tesla. Autopilot and Full Self-Driving (Supervised). URL: https://www.tesla.com/support/autopilot (visited on 08/29/2024).

[10] X. Qiang. The road to digital unfreedom: President Xi's surveillance state. *Journal of Democracy* **30**(1), 53–67, 2019.

[11] Meta AI. Introducing Llama 3.1: Our most capable models to date. URL: https://ai.meta.com/blog/meta-llama-3-1/ (visited on 08/30/2024).

[12] OpenAI. Hello GPT-4o. URL: https://openai.com/index/hello-gpt-4o/ (visited on 08/30/2024).

[13] Anthropic. Claude 3.5 Sonnet. URL: https://www.anthropic.com/news/claude-3-5-sonnet (visited on 08/30/2024).

[14] M. S. Rahaman, M. M. T. Ahsan, N. Anjum, M. M. Rahman, and M. N. Rahman. The AI Race is on! Googles Bard and OpenAIs ChatGPT Head to Head: An Opinion Article. SSRN Scholarly Paper. Rochester, NY. DOI: 10.2139/ssrn.4351785. URL: https://papers.ssrn.com/abstract=4351785 (visited on 08/30/2024). 2023.

[15] K. Roose. How ChatGPT Kicked Off an AI Arms Race. *International New York Times*, NA–NA, 2023. URL: https://go.gale.com/ps/i.do?p=AONE&sw=w&issn=22699740&v=2.1&it=r&id=GALE%7CA735981573&sid=googleScholar&linkaccess=abs (visited on 08/30/2024).

[16] Google AI. Making AI helpful for everyone. URL: https://ai.google (visited on 08/29/2024).

[17] Meta AI. Building AI experiences for everyone. URL: https://ai.meta.com (visited on 08/29/2024).

[18] OpenAI. About OpenAI. URL: https://openai.com/about/ (visited on 08/29/2024).

[19] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research* **18**(187), 1–30, 2018. URL: http://jmlr.org/papers/v18/16-456.html.

[20] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort. A White Paper on Neural Network Quantization. *CoRR*, 2021. arXiv: 2106.08295. URL: https://arxiv.org/abs/2106.08295.

[21] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-Rank Adaptation of Large Language Models. *CoRR*, 2021. DOI: 10.48550/arXiv.2106.09685. arXiv: 2106.09685. URL: http://arxiv.org/abs/2106.09685.

[22] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing Neural Networks with the Hashing Trick. *CoRR*, 2015. DOI: 10.48550/arXiv.1504.04788. arXiv: 1504.04788. URL: http://arxiv.org/abs/1504.04788.

[23] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *CoRR*, 2016. DOI: 10.48550/arXiv.1510.00149. URL: http://arxiv.org/abs/1510.00149.

[24] E. Kurtic, A. Marques, S. Pandit, M. Kurtz, and D. Alistarh. Give Me BF16 or Give Me Death? Accuracy-Performance Trade-Offs in LLM Quantization. *CoRR*, 2024. DOI: 10.48550/arXiv.2411.02355. URL: http://arxiv.org/abs/2411.02355.

[25] Y. LeCun. If you are a student interested in building the next generation of AI systems, don't work on LLMs. URL: https://x.com/ylecun/status/1793326904692428907 (visited on 08/19/2024). 2024.

[26] D. Sterratt, B. Graham, A. Gillies, and D. Willshaw. Principles of Computational Modelling in Neuroscience. Cambridge: Cambridge University Press, 2011. DOI: 10.1017/CBO9780511975899. URL: https://www.cambridge.org/core/books/principles-of-computational-modelling-in-neuroscience/9E360450B94A386051003D763A287CF0.

[27] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc Natl Acad Sci U S A* **79**(8), 2554–2558, 1982. DOI: 10.1073/pnas.79.8.2554.

[28] P. Butlin, R. Long, E. Elmoznino, Y. Bengio, J. Birch, A. Constant, G. Deane, S. M. Fleming, C. Frith, X. Ji, R. Kanai, C. Klein, G. Lindsay, M. Michel, L. Mudrik, M. A. K. Peters, E. Schwitzgebel, J. Simon, and R. VanRullen. Consciousness in Artificial Intelligence: Insights from the Science of Consciousness. *arXiv*, 2023. DOI: 10.48550/arXiv.2308.08708. URL: http://arxiv.org/abs/2308.08708.

[29] D. C. Dennett. Consciousness Explained. Penguin Books, 1991.

[30] J. L. Elman. Finding Structure in Time. *Cognitive Science* **14**(2), 179–211, 1990. DOI: https://doi.org/10.1207/s15516709cog1402_1. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1402_1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1.

[31] M. I. Jordan. Serial order: a parallel distributed processing approach. Technical report, June 1985-March 1986. (AD-A-173989/5/XAB; ICS-8604), 1986. URL: https://www.osti.gov/biblio/6910294.

[32] D. Eigen, J. Rolfe, R. Fergus, and Y. LeCun. Understanding Deep Architectures using a Recursive Convolutional Network. *CoRR*, 2014. DOI: 10.48550/arXiv.1312.1847. URL: http://arxiv.org/abs/1312.1847.

[33] P. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE* **78**(10), 1550–1560, 1990. DOI: 10.1109/5.58337. URL: https://ieeexplore.ieee.org/document/58337.

[34] S. Bae, A. Fisch, H. Harutyunyan, Z. Ji, S. Kim, and T. Schuster. Relaxed Recursive Transformers: Effective Parameter Sharing with Layer-wise LoRA. *CoRR*, 2024. URL: http://arxiv.org/abs/2410.20672.

[35] S. Herzog, C. Tetzlaff, and F. Wörgötter. Transfer entropy-based feedback improves performance in artificial neural networks. *ArXiv*, 2017. arXiv: 1706.04265 [cs.LG]. URL: https://arxiv.org/abs/1706.04265.

[36] S. Reardon. Largest brain map ever reveals fruit flys neurons in exquisite detail. *Nature*, 2024. DOI: 10.1038/d41586-024-03190-y. URL: https://www.nature.com/articles/d41586-024-03190-y.

[37] S. A. Valizadeh, F. Liem, S. Mérillat, J. Hänggi, and L. Jäncke. Identification of individual subjects on the basis of their brain anatomical features. *Scientific*

Reports **8**(5611), 2018. DOI: 10.1038/s41598-018-23696-6. URL: https://www.nature.com/articles/s41598-018-23696-6.

[38] N. T. Markov, J. Vezoli, P. Chameau, A. Falchier, R. Quilodran, C. Huissoud, C. Lamy, P. Misery, P. Giroud, S. Ullman, P. Barone, C. Dehay, K. Knoblauch, and H. Kennedy. Anatomy of hierarchy: feedforward and feedback pathways in macaque visual cortex. *The Journal of Comparative Neurology* **522**(1), 225–259, 2014. DOI: 10.1002/cne.23458.

[39] A. Shapson-Coe et al. A petavoxel fragment of human cerebral cortex reconstructed at nanoscale resolution. *Science* **384**(6696), eadk4858, 2024. DOI: 10.1126/science.adk4858. URL: https://www.science.org/doi/10.1126/science.adk4858.

[40] S. M. Bohte, J. N. Kok, and H. La Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing* **48**(1), 17–37, 2002. DOI: https://doi.org/10.1016/S0925-2312(01)00658-0. URL: https://www.sciencedirect.com/science/article/pii/S0925231201006580.

[41] Y. Bengio, D.-H. Lee, J. Bornschein, and Z. Lin. Towards Biologically Plausible Deep Learning. *ArXiv*, 2015. DOI: 10.48550/arXiv.1502.04156. URL: https://api.semanticscholar.org/CorpusID:13068979.

[42] A. H. Marblestone, G. Wayne, and K. P. Kording. Toward an Integration of Deep Learning and Neuroscience. *Frontiers in Computational Neuroscience* **10**, 2016. DOI: 10.3389/fncom.2016.00094. URL: https://www.frontiersin.org/journals/computational-neuroscience/articles/10.3389/fncom.2016.00094/full.

[43] Y. LeCun, Y. Bengio, and G. Hinton. Deep Learning. *Nature* **521**, 436–44, 2015. DOI: 10.1038/nature14539.

[44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. *CoRR*, 2023. DOI: 10.48550/arXiv.1706.03762. URL: http://arxiv.org/abs/1706.03762.

[45] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *CoRR*, 2021. URL: http://arxiv.org/abs/2005.11401.

[46] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *CoRR*, 2023. DOI: 10.48550/arXiv.2201.11903. URL: http://arxiv.org/abs/2201.11903.

[47] J.-S. Byun, J. Chun, J. Kil, and A. Perrault. ARES: Alternating Reinforcement Learning and Supervised Fine-Tuning for Enhanced Multi-Modal Chain-of-Thought Reasoning Through Diverse AI Feedback. *CoRR*, 2024. DOI: 10.48550/arXiv.2407.00087. URL: http://arxiv.org/abs/2407.00087.

[48] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature* **323**, 533–536, 1986. URL: https://www.nature.com/articles/323533a0.

[49] N. Kriegeskorte. Deep Neural Networks: A New Framework for Modeling Biological Vision and Brain Information Processing. *Annual Review of Vision Science* **1**(Volume 1, 2015), 417–446, 2015. DOI: 10.1146/annurev-vision-082114-035447. URL: https://www.annualreviews.org/content/journals/10.1146/annurev-vision-082114-035447.

[50] S.-i. Amari. A Theory of Adaptive Pattern Classifiers. *IEEE Transactions on Electronic Computers* **EC-16**(3), 299–307, 1967. DOI: 10.1109/PGEC.1967.264666. URL: https://ieeexplore.ieee.org/abstract/document/4039068.

[51] S.-i. Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing* **5**(4), 185–196, 1993. DOI: 10.1016/0925-2312(93)90006-O. URL: https://www.sciencedirect.com/science/article/pii/092523129390006O.

[52] S.-i. Amari. Natural Gradient Works Efficiently in Learning. *Neural Computation* **10**(2), 251–276, 1998. DOI: 10.1162/089976698300017746. URL: https://ieeexplore.ieee.org/abstract/document/6790500.

[53] H. J. Kelley. Gradient Theory of Optimal Flight Paths. *ARS Journal* **30**, 947–954, 1960. URL: https://arc.aiaa.org/doi/10.2514/8.5282.

[54] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* **65**(6), 386–408, 1958. DOI: 10.1037/h0042519. URL: https://doi.apa.org/doi/10.1037/h0042519.

[55] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *CoRR*, 2017. DOI: 10.48550/arXiv.1412.6980. URL: http://arxiv.org/abs/1412.6980.

[56] G. Hinton, N. Srivastava, and K. Swersky. Neural Networks for Machine Learning; Lecture 6b, A bag of tricks for mini-batch gradient descent. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (visited on 09/22/2024). 2014.

[57] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86**(11), 2278–2324, 1998. DOI: 10.1109/5.726791. URL: https://ieeexplore.ieee.org/document/726791.

[58] J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation. *CoRR*, 2015. DOI: 10.48550/arXiv.1411.4038. URL: http://arxiv.org/abs/1411.4038.

[59] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *arXiv.org*, 2015. URL: https://arxiv.org/abs/1505.04597v1.

[60] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. *CoRR*, 2022. DOI: 10.48550/arXiv.1312.6114. URL: http://arxiv.org/abs/1312.6114.

[61] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. *arXiv.org*, 2017. URL: https://arxiv.org/abs/1706.03762v7.

[62] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv.org*, 2015. URL: https://arxiv.org/abs/1512.03385v1.

[63] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely Connected Convolutional Networks. *CoRR*, 2018. DOI: 10.48550/arXiv.1608.06993. URL: http://arxiv.org/abs/1608.06993.

[64] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks* **5**(2), 157–166, 1994. DOI: 10.1109/72.279181. URL: https://ieeexplore.ieee.org/document/279181.

[65] S. Hochreiter and J. Schmidhuber. Long Short-term Memory. *Neural computation* **9**, 1735–80, 1997. DOI: 10.1162/neco.1997.9.8.1735.

[66] G. Van Rossum and F. L. Drake Jr. Python reference manual. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[67] N. D. Matsakis and F. S. Klock II. The rust language. *ACM SIGAda Ada Letters* **34**. (3). ACM, 103–104, 2014.

[68] J. Ansel et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)* ACM, 2024. DOI: 10.1145/3620665.3640366. URL: https://pytorch.org/assets/pytorch2-2.pdf.

[69] Martín Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org. URL: https://www.tensorflow.org/. 2015.

[70] J. Stone and N. M. et al. Rayon: A data parallelism library for Rust. Version 1.10.0. URL: https://github.com/rayon-rs/rayon. 2024.

[71] I. GitHub. GitHub Copilot: The AI coding assistant elevating developer workflows. Online; AI writing assistance tool. URL: https://github.com/features/copilot. 2024.

[72] OpenAI. ChatGPT: Large Language Model. Online; AI language model by OpenAI. Model: gpt-4o-2024-08-06. URL: https://platform.openai.com/docs/models/#gpt-4o. 2024.

[73] OpenAI. ChatGPT: Large Language Model. Online; AI language model by OpenAI. Model: gpt-4o-mini-2024-07-18. URL: https://platform.openai.com/docs/models/#gpt-4o. 2024.

[74] Anthropic. Claude 3.5 Sonnet: Large Language Model. Online; AI language model by Anthropic. Model: claude-3-5-sonnet-20240620. URL: https://docs.anthropic.com/en/docs/about-claude/models#model-comparison-table. 2024.

[75] Anthropic. Claude 3.5 Sonnet (New): Large Language Model. Online; AI language model by Anthropic. Model: claude-3-5-sonnet-20241022. URL: https://docs.anthropic.com/en/docs/about-claude/models#model-comparison-table. 2024.

[76] D. Masters and C. Luschi. Revisiting Small Batch Training for Deep Neural Networks. *CoRR*, 2018. arXiv: 1804.07612. URL: http://arxiv.org/abs/1804.07612.

[77] I. Grammarly. Grammarly: AI-Powered Writing Assistant. Online; AI writing assistance tool. URL: https://www.grammarly.com/. 2024.

[78] R. A. Fisher. Iris. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C56C76. 1936.

[79] H. Fanaee-T. Bike Sharing. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5W894. 2013.

[80] K. A. Kristoffersen, K. H. Liland, U. Böcker, S. G. Wubshet, D. Lindberg, S. J. Horn, and N. K. Afseth. FTIR-based hierarchical modeling for prediction of average molecular weights of protein hydrolysates. *Talanta* **205**, 120084, 2019. DOI: https://doi.org/10.1016/j.talanta.2019.06.084. URL: https://www.sciencedirect.com/science/article/pii/S003991401930709X.

[81] R. Helin, U. G. Indahl, O. Tomic, and K. H. Liland. On the possible benefits of deep learning for spectral preprocessing. *Journal of Chemometrics* **36**(2), e3374, 2022. DOI: https://doi.org/10.1002/cem.3374. eprint: https://analyticalsciencejournals.onlinelibrary.wiley.com/doi/pdf/10.1002/cem.3374. URL: https://analyticalsciencejournals.onlinelibrary.wiley.com/doi/abs/10.1002/cem.3374.

[82] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *University of Toronto*, 2012.

[83] Meta AI. Introducing Meta Llama 3: The most capable openly available LLM to date. URL: https://ai.meta.com/blog/meta-llama-3/ (visited on 12/11/2024).

A Tables of software

PACKAGE	VERSION	PURPOSE
`neurons`	2.6.2	Modular neural networks [4].

Table A.1: Rust package used. Notably, `Rayon` [70] is the only external package used within `neurons`, but omitted from this table as the latter is its own package.

PACKAGE	VERSION	PURPOSE
`torch`	2.3.0	Validation of our framework.
`matplotlib`	3.8.0	Visualisation of training metrics.
`numpy`	1.26.3	Statistical calculation.

Table A.2: Python packages used. These are used for validation of our `neurons` framework, and preparation and visualisation of results.

B | Probed metrics

MODEL	ORIGINAL Accuracy Loss	SKIP OFF Accuracy Loss	FEEDBACK OFF Accuracy Loss
FFN w. bypassing skip	0.9333 ± 0.0577 0.1872 ± 0.0727	0.4667 ± 0.2625 5.4459 ± 2.2566	
COMBINE x2 w. bypassing skip	0.7000 ± 0.3000 1.2226 ± 1.2143	0.4267 ± 0.1937 5.6673 ± 3.0591	0.2600 ± 0.0879 8.1607 ± 3.0011
COMBINE x3 w. bypassing skip	0.9667 ± 0.0000 0.2798 ± 0.0451	0.7067 ± 0.2015 3.3793 ± 2.7378	0.5533 ± 0.1343 5.2767 ± 1.6348
COMBINE x4 w. bypassing skip	0.9667 ± 0.0000 0.4612 ± 0.0000	0.7133 ± 0.1746 2.2757 ± 1.7168	0.2867 ± 0.1343 9.0565 ± 1.4727
COUPLE x2 w. bypassing skip	0.9444 ± 0.0416 0.3191 ± 0.2258	0.7733 ± 0.1718 2.1884 ± 1.9317	0.2333 ± 0.1033 9.0348 ± 1.6492
COUPLE x3 w. bypassing skip	0.8444 ± 0.1257 0.8053 ± 0.8358	0.7600 ± 0.1948 2.3039 ± 2.5308	0.3867 ± 0.2197 8.1946 ± 2.8647
COUPLE x4 w. bypassing skip	0.9250 ± 0.0924 0.8299 ± 1.1558	0.9133 ± 0.0806 1.0661 ± 1.1761	0.3533 ± 0.1275 6.6869 ± 1.7476
FFN	0.9067 ± 0.1062 0.1601 ± 0.1384		
COMBINE x2	0.8750 ± 0.1605 0.4054 ± 0.4785		0.2800 ± 0.1002 8.5336 ± 2.0082
COMBINE x3	0.9333 ± 0.0000 0.2025 ± 0.0640		0.3867 ± 0.2197 7.5687 ± 3.3898
COMBINE x4	0.7667 ± 0.2000 2.8355 ± 2.7446		0.3333 ± 0.1687 7.7884 ± 1.3712
COUPLE x2	0.9000 ± 0.1225 0.2365 ± 0.2388		0.2867 ± 0.1668 8.5482 ± 0.6189
COUPLE x3	0.9000 ± 0.0981 0.3677 ± 0.2724		0.5000 ± 0.1687 6.8562 ± 2.4192
COUPLE x4	0.9417 ± 0.0493 0.3802 ± 0.3442		0.5000 ± 0.1520 6.6794 ± 2.1662

Table B.1: Probed results of IRIS for classification. Cross-entropy loss. The mean and standard deviation are calculated over five runs.

MODEL	ORIGINAL Accuracy Loss	SKIP OFF Accuracy Loss	FEEDBACK OFF Accuracy Loss
FFN w. bypassing skip	0.9846 ± 0.0041 0.0788 ± 0.0131	0.9977 ± 0.0028 0.0028 ± 0.0025	
COMBINE x2 w. bypassing skip	0.9626 ± 0.0101 0.2520 ± 0.0877	0.9793 ± 0.0148 0.0725 ± 0.0547	0.1425 ± 0.0935 5.2835 ± 0.7749
COMBINE x3 w. bypassing skip	0.9352 ± 0.0204 0.8481 ± 0.2875	0.9471 ± 0.0067 0.6425 ± 0.0996	0.3046 ± 0.0717 3.4350 ± 0.5968
COUPLE x2 w. bypassing skip	0.9714 ± 0.0081 0.2423 ± 0.0552	0.9897 ± 0.0067 0.0423 ± 0.0259	0.0276 ± 0.0245 3.6988 ± 0.1918
COUPLE x3 w. bypassing skip	0.8242 ± 0.0628 2.1106 ± 0.7940	0.8034 ± 0.0794 2.2961 ± 1.0132	0.0230 ± 0.0163 3.5176 ± 0.2149
FFN	0.9857 ± 0.0056 0.0796 ± 0.0305		
COMBINE x2	0.9659 ± 0.0153 0.2105 ± 0.0963		0.1701 ± 0.0765 3.6336 ± 0.5338
COMBINE x3	0.8604 ± 0.1148 1.8452 ± 1.6071		0.2816 ± 0.1055 4.2700 ± 1.6590
COUPLE x2	0.9659 ± 0.0064 0.2663 ± 0.0731		0.0563 ± 0.0816 3.6007 ± 0.2767
COUPLE x3	0.8036 ± 0.0377 2.1953 ± 0.5950		0.0494 ± 0.0194 3.4576 ± 0.1840

Table B.2: Probed results of FTIR dense for classification. Cross-entropy loss. The mean and standard deviation are calculated over five runs.

MODEL	ORIGINAL Accuracy Loss	SKIP OFF Accuracy Loss	FEEDBACK OFF Accuracy Loss
FFN w. bypassing skip	0.9780 ± 0.0055 0.1842 ± 0.0824	0.9920 ± 0.0046 0.0392 ± 0.0256	
COMBINE x2 w. bypassing skip	0.9121 ± 0.0604 0.7358 ± 0.6316	0.9621 ± 0.0425 0.3167 ± 0.4536	0.5747 ± 0.2902 4.3425 ± 3.8608
COMBINE x3 w. bypassing skip	0.8425 ± 0.1723 0.5985 ± 0.4549	0.9161 ± 0.1338 0.3156 ± 0.4050	0.3770 ± 0.3241 3.1353 ± 2.6770
COUPLE x2 w. bypassing skip	0.9615 ± 0.0000 0.2245 ± 0.0000	0.9839 ± 0.0111 0.0641 ± 0.0491	0.4011 ± 0.3155 4.1851 ± 3.3476
COUPLE x3 w. bypassing skip	0.9725 ± 0.0000 0.1814 ± 0.0000	0.9736 ± 0.0134 0.1677 ± 0.0972	0.2609 ± 0.1850 3.1678 ± 0.2361
FFN	0.9780 ± 0.0000 0.2360 ± 0.0019		
COMBINE x2	0.6662 ± 0.3952 1.1551 ± 1.3111		0.3506 ± 0.1860 2.4749 ± 0.7981
COMBINE x3	0.9643 ± 0.0143 0.2544 ± 0.0805		0.5023 ± 0.3557 3.4334 ± 3.7490
COUPLE x2	0.9780 ± 0.0000 0.2831 ± 0.0000		0.2414 ± 0.1895 2.9122 ± 0.4641
COUPLE x3	0.9725 ± 0.0000 0.2137 ± 0.0000		0.0644 ± 0.0834 3.3675 ± 0.2097

Table B.3: Probed results of FTIR convolutional for classification. Cross-entropy loss. The mean and standard deviation are calculated over five runs.

MODEL	**ORIGINAL** Loss	**SKIP OFF** Loss	**FEEDBACK OFF** Loss
FFN w. bypassing skip	286.2516 ± 2.3320	329.5180 ± 3.1683	
COMBINE x2 w. bypassing skip	295.0743 ± 35.3704	331.5135 ± 25.5780	2.45e+03 ± 15.1166
COMBINE x3 w. bypassing skip	301.7532 ± 37.8552	333.4908 ± 31.8292	2.47e+03 ± 5.2111
COUPLE x2 w. bypassing skip	242.2010 ± 4.6346	278.5039 ± 3.3245	2.46e+03 ± 3.3482
COUPLE x3 w. bypassing skip	286.3460 ± 13.3056	321.8900 ± 13.1515	2.48e+03 ± 0.6274
FFN	289.4797 ± 3.7274		
COMBINE x2	279.8956 ± 26.8177		2.45e+03 ± 7.3723
COMBINE x3	303.6938 ± 27.8487		2.46e+03 ± 6.9724
COUPLE x2	261.5397 ± 40.8263		2.47e+03 ± 3.2956
COUPLE x3	250.9078 ± 32.2835		2.48e+03 ± 0.7161

Table B.4: Probed results of FTIR dense for regression. RMSE loss. The mean and standard deviation are calculated over five runs.

MODEL	**ORIGINAL** Loss	**SKIP OFF** Loss	**FEEDBACK OFF** Loss
FFN w. bypassing skip	482.5231 ± 8.7768	497.5622 ± 13.7966	
COMBINE x2 w. bypassing skip	505.7155 ± 135.8955	505.7958 ± 120.9623	2.31e+03 ± 112.8095
COMBINE x3 w. bypassing skip	2.04e+08 ± 4.09e+08	2.62e+08 ± 5.23e+08	3.01e+03 ± 1.49e+03
COUPLE x2 w. bypassing skip	364.5621 ± 21.2237	390.4704 ± 8.0995	2.44e+03 ± 6.7537
COUPLE x3 w. bypassing skip	324.5447 ± 27.6966	367.1906 ± 19.5865	2.47e+03 ± 0.9461
FFN	483.8680 ± 14.9439		
COMBINE x2	372.0797 ± 28.1221		2.42e+03 ± 7.5351
COMBINE x3	551.3221 ± 268.0685		2.43e+03 ± 83.2575
COUPLE x2	345.7501 ± 15.0615		2.44e+03 ± 9.2476
COUPLE x3	323.9352 ± 9.9574		2.47e+03 ± 1.1681

Table B.5: Probed results of FTIR convolutional for regression. RMSE loss. The mean and standard deviation are calculated over five runs.

MODEL	ORIGINAL Loss	SKIP OFF Loss	FEEDBACK OFF Loss
FFN w. bypassing skip	34.4340 ± 3.2559	57.0422 ± 14.4297	
COMBINE x2 w. bypassing skip	90.4548 ± 9.2778	176.5325 ± 48.5754	199.3936 ± 111.4051
COMBINE x3 w. bypassing skip	90.8480 ± 2.4589	1.30e+03 ± 2.16e+03	934.7963 ± 1.22e+03
COMBINE x4 w. bypassing skip	86.0871 ± 4.1763	200.9613 ± 23.0981	145.7140 ± 90.6394
COUPLE x2 w. bypassing skip	34.0238 ± 0.6375	35.1190 ± 1.2446	167.9949 ± 13.0550
COUPLE x3 w. bypassing skip	35.1117 ± 2.2704	39.0023 ± 2.2510	160.3087 ± 20.7902
COUPLE x4 w. bypassing skip	35.9820 ± 2.9379	44.3923 ± 4.9732	163.1632 ± 10.4609
FFN	33.8490 ± 1.2388		
COMBINE x2	131.0181 ± 41.6974		157.7652 ± 32.1280
COMBINE x3	109.3255 ± 23.2035		2.02e+03 ± 3.55e+03
COMBINE x4	113.6105 ± 9.7376		498.6023 ± 504.6533
COUPLE x2	33.9515 ± 1.9739		169.6966 ± 15.3532
COUPLE x3	34.7950 ± 2.2272		169.4444 ± 13.0809
COUPLE x4	38.2286 ± 3.3478		177.8111 ± 5.0561

Table B.6: Probed results of BIKE for regression. RMSE loss. The mean and standard deviation are calculated over five runs.

C | Wall clock times

MODEL	TRAIN	VALIDATE
FFN w. bypassing skip	2.4258 ± 0.1465	0.2394 ± 0.0187
COMBINE x2 w. bypassing skip	2.2012 ± 0.0712	0.2732 ± 0.0210
COMBINE x3 w. bypassing skip	2.2062 ± 0.0561	0.3321 ± 0.0097
COMBINE x4 w. bypassing skip	2.3063 ± 0.0461	0.3963 ± 0.0197
COUPLE x2 w. bypassing skip	4.5922 ± 0.0722	0.2428 ± 0.0186
COUPLE x3 w. bypassing skip	6.5030 ± 0.0815	0.2856 ± 0.0143
COUPLE x4 w. bypassing skip	8.5986 ± 0.0938	0.3304 ± 0.0130
FFN	2.2339 ± 0.2867	0.2079 ± 0.0360
COMBINE x2	2.3942 ± 0.1326	0.3455 ± 0.0648
COMBINE x3	2.1433 ± 0.0550	0.3401 ± 0.0138
COMBINE x4	2.2884 ± 0.0355	0.3991 ± 0.0266
COUPLE x2	4.6079 ± 0.1417	0.2350 ± 0.0105
COUPLE x3	6.5708 ± 0.0895	0.2623 ± 0.0134
COUPLE x4	8.5289 ± 0.1022	0.3245 ± 0.0060

Table C.1: Time differences of IRIS models for classification. All times are in milliseconds. The mean and standard deviation are calculated over five runs. The training times are obtained from one epoch.

MODEL	TRAIN	VALIDATE
FFN w. bypassing skip	114.9545 ± 1.4908	17.0450 ± 0.4543
COMBINE x2 w. bypassing skip	121.7461 ± 0.6204	21.6945 ± 0.2621
COMBINE x3 w. bypassing skip	127.4372 ± 0.2882	28.0633 ± 0.8674
COUPLE x2 w. bypassing skip	165.8079 ± 0.8126	21.5225 ± 0.4541
COUPLE x3 w. bypassing skip	224.6669 ± 0.5749	26.5877 ± 0.8760
FFN	116.3228 ± 2.1466	16.7336 ± 0.5237
COMBINE x2	121.6331 ± 0.6375	21.7531 ± 0.3461
COMBINE x3	132.4371 ± 6.0043	29.4469 ± 2.6525
COUPLE x2	168.1891 ± 3.5899	21.2200 ± 0.4854
COUPLE x3	224.0162 ± 0.7040	26.0266 ± 0.9249

Table C.2: Time differences of FTIR dense models for classification. All times are in milliseconds. The mean and standard deviation are calculated over five runs. The training times are obtained from one epoch.

MODEL	TRAIN	VALIDATE
FFN w. bypassing skip	29.3768 ± 2.4873	6.9509 ± 0.0537
COMBINE x2 w. bypassing skip	33.4597 ± 0.4914	11.9801 ± 0.3650
COMBINE x3 w. bypassing skip	37.6385 ± 0.5017	17.3773 ± 0.6920
COUPLE x2 w. bypassing skip	40.1032 ± 0.7115	11.7751 ± 0.0989
COUPLE x3 w. bypassing skip	51.2194 ± 0.3787	16.4428 ± 0.4318
FFN	29.0975 ± 2.2787	7.0207 ± 0.3973
COMBINE x2	32.6733 ± 0.4177	11.9375 ± 0.0946
COMBINE x3	39.4026 ± 4.3102	17.2540 ± 0.4477
COUPLE x2	47.6980 ± 0.8026	13.2830 ± 0.9639
COUPLE x3	51.0296 ± 0.5958	16.7820 ± 0.3534

Table C.3: Time differences of FTIR convolutional models for classification. All times are in milliseconds. The mean and standard deviation are calculated over five runs. The training times are obtained from one epoch.

MODEL	TRAIN	VALIDATE
FFN w. bypassing skip	113.2124 ± 0.2437	16.6663 ± 0.1616
COMBINE x2 w. bypassing skip	119.9042 ± 1.4418	21.9506 ± 0.5826
COMBINE x3 w. bypassing skip	124.5665 ± 0.6768	27.2488 ± 1.0281
COUPLE x2 w. bypassing skip	162.3222 ± 0.8764	21.4442 ± 0.9664
COUPLE x3 w. bypassing skip	220.0654 ± 0.7863	26.1817 ± 0.6371
FFN	131.9762 ± 38.6925	16.4012 ± 0.2023
COMBINE x2	118.3726 ± 0.4569	21.7252 ± 0.6763
COMBINE x3	124.5314 ± 0.5288	27.5524 ± 0.9030
COUPLE x2	162.4628 ± 0.2674	20.7092 ± 0.1797
COUPLE x3	220.6357 ± 0.3985	26.3073 ± 0.3682

Table C.4: Time differences of FTIR dense models for regression. All times are in milliseconds. The mean and standard deviation are calculated over five runs. The training times are obtained from one epoch.

MODEL	TRAIN	VALIDATE
FFN w. bypassing skip	27.4214 ± 0.5171	6.9862 ± 0.1419
COMBINE x2 w. bypassing skip	32.4999 ± 0.6110	12.2905 ± 0.3568
COMBINE x3 w. bypassing skip	39.4587 ± 2.1134	18.2817 ± 0.6578
COUPLE x2 w. bypassing skip	39.1309 ± 0.4757	11.7469 ± 0.1154
COUPLE x3 w. bypassing skip	50.4408 ± 0.2679	16.7358 ± 0.5162
FFN	27.0118 ± 0.3070	6.7585 ± 0.0280
COMBINE x2	31.9988 ± 0.2372	12.0675 ± 0.1104
COMBINE x3	37.0163 ± 0.6114	17.4675 ± 0.3859
COUPLE x2	42.2645 ± 2.8125	12.0460 ± 0.5635
COUPLE x3	51.1265 ± 1.7764	16.5999 ± 0.4260

Table C.5: Time differences of FTIR convolutional models for regression. All times are in milliseconds. The mean and standard deviation are calculated over five runs. The training times are obtained from one epoch.

MODEL	TRAIN	VALIDATE
FFN w. bypassing skip	75.4366 ± 0.3417	7.9148 ± 0.4483
COMBINE x2 w. bypassing skip	81.1277 ± 0.4414	12.6288 ± 0.7037
COMBINE x3 w. bypassing skip	85.4789 ± 0.5997	15.4259 ± 0.3808
COMBINE x4 w. bypassing skip	90.1427 ± 0.9974	18.6437 ± 0.4425
COUPLE x2 w. bypassing skip	132.6597 ± 0.2225	10.2731 ± 1.3460
COUPLE x3 w. bypassing skip	181.2799 ± 3.6450	11.4561 ± 0.6390
COUPLE x4 w. bypassing skip	221.7355 ± 2.1848	13.9325 ± 1.3689
FFN	75.8309 ± 3.6649	7.6780 ± 0.2954
COMBINE x2	79.7929 ± 0.6381	12.2645 ± 1.1217
COMBINE x3	82.7786 ± 1.3884	14.8905 ± 0.4632
COMBINE x4	88.7061 ± 0.9816	17.6067 ± 0.8124
COUPLE x2	131.4721 ± 0.6503	9.4559 ± 0.3344
COUPLE x3	179.1457 ± 1.9729	11.8176 ± 1.3345
COUPLE x4	223.0262 ± 2.0298	12.8108 ± 0.4547

Table C.6: Time differences of BIKE models for regression. All times are in milliseconds. The mean and standard deviation are calculated over five runs. The training times are obtained from one epoch.

D | Time-weighted validation

For our time-weighted curves, we omit quantile ranges to better distinguish between the models.

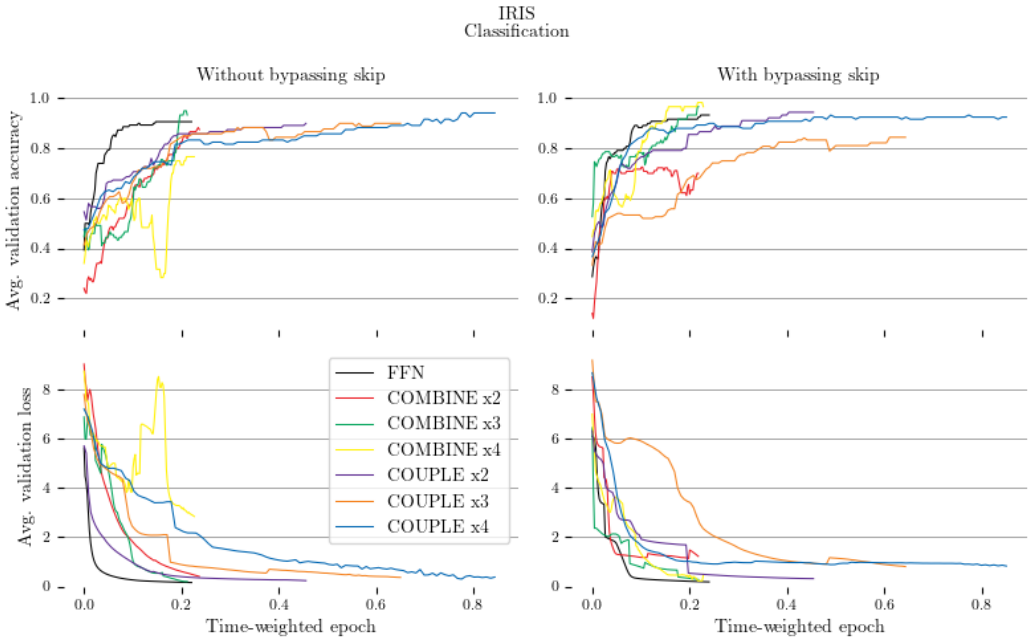


Figure D.1: Weighted iris classification validation metrics from training. Cross-entropy loss.

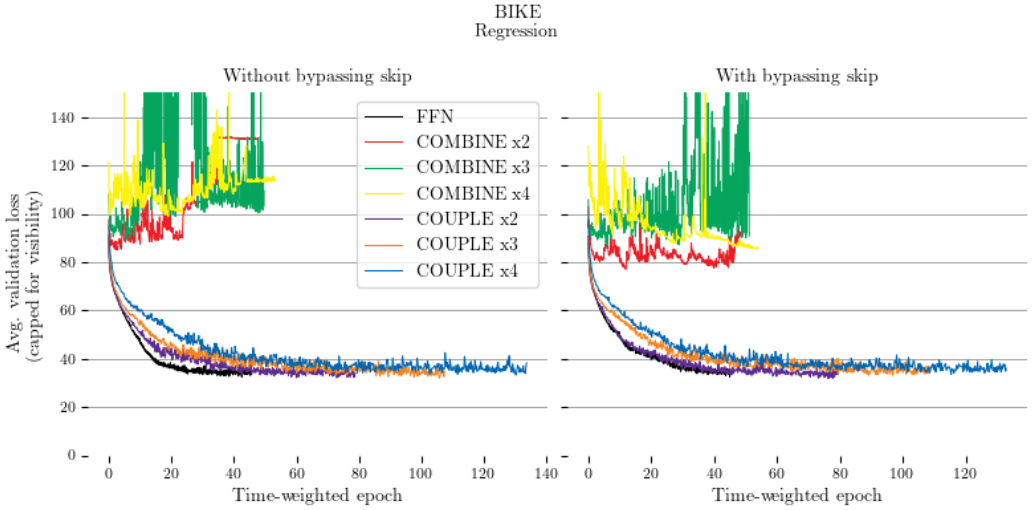


Figure D.2: Weighted bike-sharing regression validation metrics from training. RMSE loss.

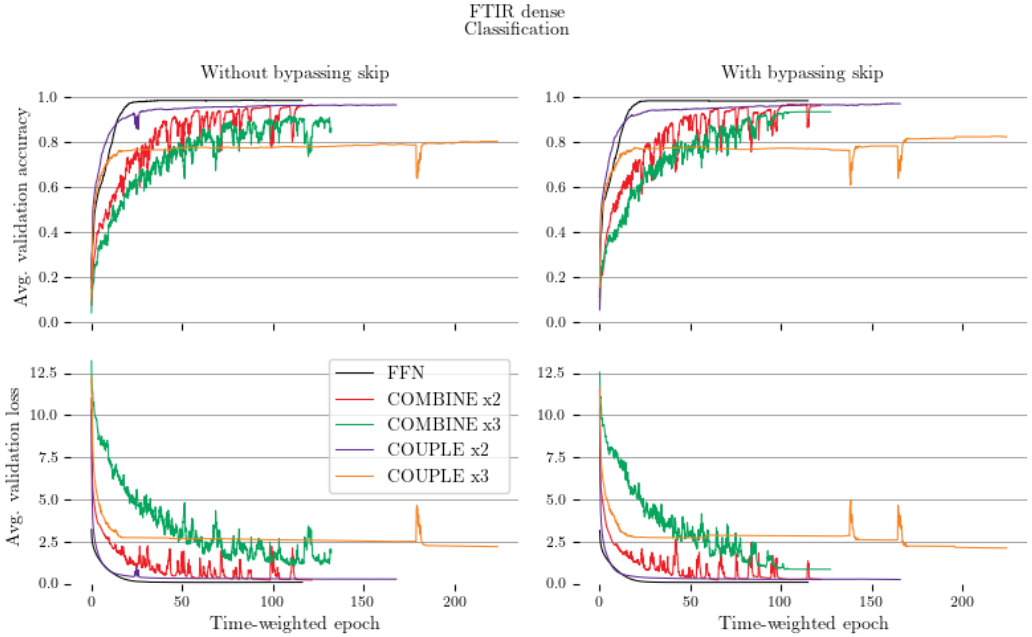


Figure D.3: Weighted dense FTIR classification validation metrics from training. Cross-entropy loss.

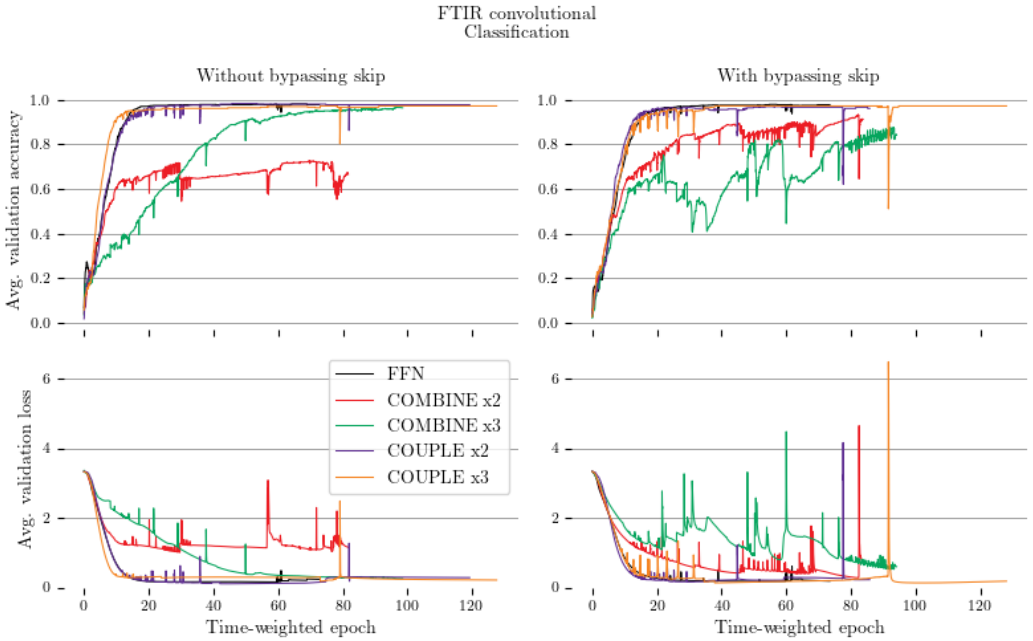


Figure D.4: Weighted convolutional FTIR classification validation metrics from training. Cross-entropy loss.

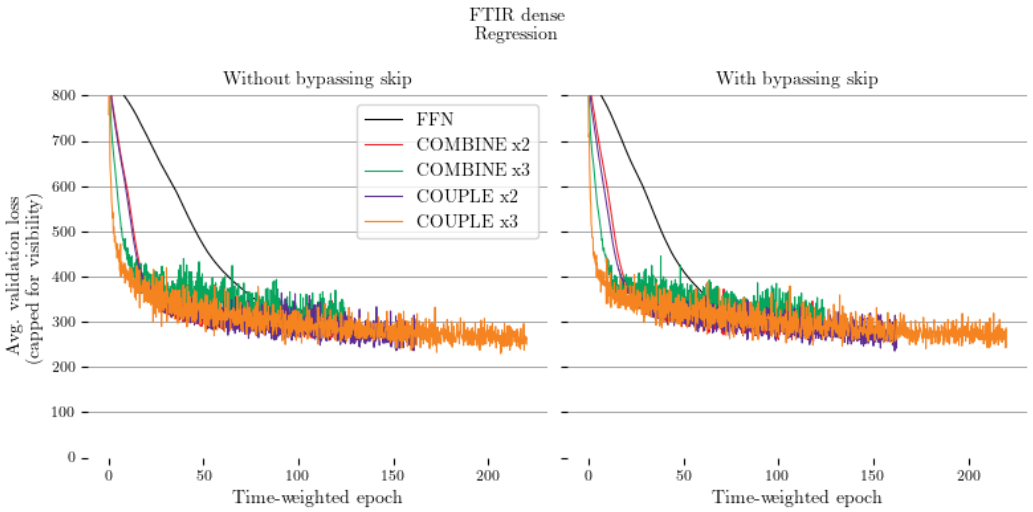


Figure D.5: Weighted dense FTIR regression validation metrics from training. RMSE loss.

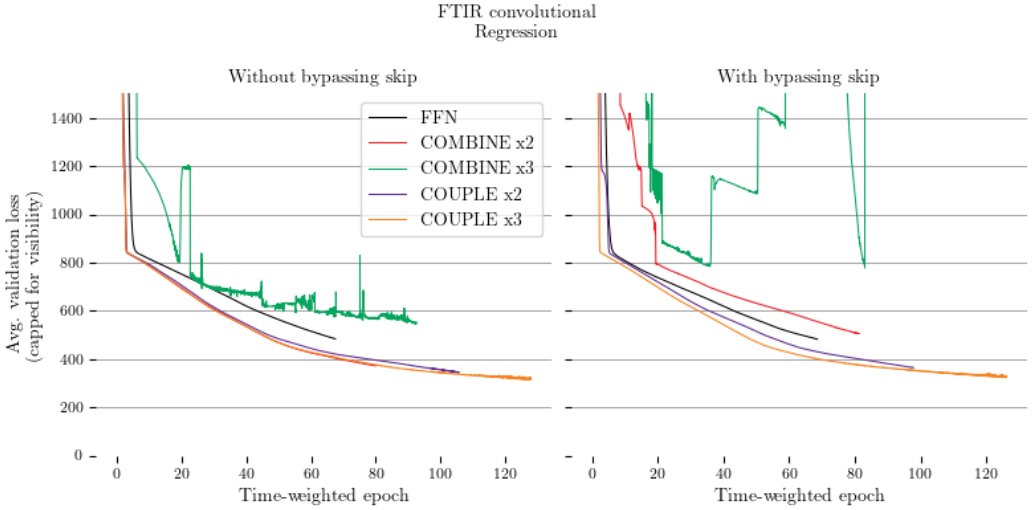
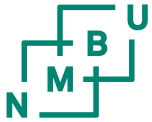


Figure D.6: Weighted convolutional FTIR regression validation metrics from training. RMSE loss.



Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskaplege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway