Chapter 2

Background and Preliminaries

In this chapter we will start by introducing some of the most fundamental concepts and results in deep learning (DL) that will be relevant to the work in this thesis. We will then cover predictive coding and energy-based models (EBMs). Specifically, we will introduce the EBM framework by studying Hopfield networks, we will then formalise discriminative and generative predictive coding networks (PCNs) with their corresponding energy-based interpretations. Finally, we will conclude this chapter with a thorough introduction to reinforcement learning (RL), starting from the fundamentals to important algorithms and results.

2.1 Artificial Neural Networks

The term *neural network* broadly captures most popular deep learning models, all of which have been developed as a result of a culmination of centuries of progress in the problem of function approximation [4, p. 217]. Some fundamental successes in history that have contributed to the popularity of neural networks include: the very first mathematical model of the biological neuron [47], the perceptron learning algorithm [6], efficient implementations of the chain rule of calculus [12] and the successful training of deep networks based on the contrastive divergence procedure [48]. Until recently, support vector machines (SVMs)

[49] were the go-to model for classification tasks; they are easier to train and tune since their objective function is convex. However, after over a decade of successes it has become clear that neural networks are here to stay, and with advances in optimization techniques and design of learning algorithms, we expect to see neural networks continue to fulfill their potential [4, p. 220].

The goal of neural networks is to approximate some function $y = f^*(\mathbf{x})$ that maps an input $\mathbf{x}$ to an output y. Neural networks consist of layers of *neurons* or *units* that process information from input to output, typically in one direction. Each neuron has an associated set of *weights* and *biases* meant to model the neuron's firing rate given some input. Together all these weights and biases parameterise the neural network, typically these parameters are denoted by $\boldsymbol{\theta}$. So a neural network parameterised by $\boldsymbol{\theta}$ defines a mapping $y = f(\mathbf{x}; \boldsymbol{\theta})$. The ability for the strength of synaptic connections in the brain to change is referred to as *synaptic plasticity*, and parameter updates to $\boldsymbol{\theta}$ adjusted by learning algorithms correspond to this ability [14, p. 379]. In the end we wish to learn the set of parameters $\boldsymbol{\theta}^*$ that best approximate the function f^*. Networks of this form are typically referred to as feed-forward neural networks since their computation can be represented as a *directed acyclic graph* (DAG). Networks with loops or feedback connections are called *recurrent neural networks* (RNNs) and are beyond the scope of this work [4, Ch. 10].

Many different types of layers exist in the DL literature and the particular choice and ordering of layers is referred to as the *architecture* of a neural network. Most layers are typically followed by a non-linear *activation function* g that models the spiking or firing of a neuron in the brain. In theory, the use of non-linear activation functions allows neural networks to capture complex non-linear relationships; if training is done correctly this makes them powerful models for many machine learning tasks. The choice of which activation function to use is important, for example, the adoption of the *rectified linear unit* (ReLU) activation function over the sigmoid function is a major landmark in the DL literature [4, p. 219]. This choice was actually biologically motivated since the ReLU function $f(\mathbf{x}) = \mathsf{max}(0, \mathbf{x})$ captures two important properties of biological neurons: most of the time biological neurons are inactive, and for some inputs biological neurons fire at a rate proportional to their input [50]. While it seems much of the success related to neural networks is biologically inspired, it is important to note that neural networks are not intended to be realistic models of the brain, but are rather loosely inspired by results and hypothesis related to neural activity, brain function and learning in the brain.

In this section, we will start by covering the fundamental building blocks that make up neural networks, before describing gradient based optimisation techniques and the backpropagation algorithm. We will then conclude this section by outlining the basic operations of *convolutional neural networks* (CNNs), which will be relevant for some of the experiments in this thesis.

2.1.1 The Perceptron

The perceptron [6] is a learning algorithm for classifying inputs in some vector space into one of two distinct classes $\{-1, +1\}$. The perceptron is based on the McCulloch-Pitts neuron [47] - a simple mathematical model which approximates the neural activity of a neuron by a weighted sum of its inputs passed through a threshold function [51, p. 569],

$$y = \mathbb{1}\left(\sum_{i=1} w_i \cdot x_i > \lambda\right) \tag{2.1}$$

for some threshold λ. Here each x_i corresponds to an input feature and each w_i corresponds to its associated weight. The perceptron extends this idea by adding a bias term b and replacing the threshold function with the sign function. The *pre-activation* value z is computed by taking a linear combination of the inputs $x_1, ..., x_D$ plus the bias term b. The output value or prediction $\hat{y}$ (sometimes called the *activation* and denoted a) is computed by passing z through the sign function, classifying it into one of $\{-1, +1\}$, see Figure 2.1 for a visual representation of this computation.

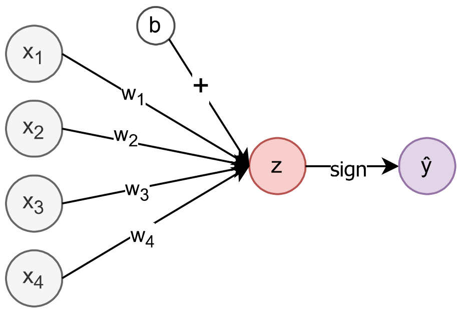

Figure 2.1: Perceptron with input dimension $D = 4$.

In machine learning tasks like classification, we are typically given a dataset of input output-pairs, $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{n}$, the goal is to learn the parameters of the model that best match the dataset. Algorithm 1 presents the original perceptron algorithm [6] for classifying some given dataset $\mathcal{D}$. The perceptron algorithm can be applied to a rigid dataset (fixed size) or it can be applied in an online fashion (to infinity and beyond). Either way, the perceptron algorithm has been show to maintain nice convergence guarantees in both these settings [52, 53].

```
Algorithm 1 The Perceptron Algorithm
Input: Fixed size dataset $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{n}$, or Online $\langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{\infty}$
Initialize: $\mathbf{w_1} = \mathbf{0} \in \mathbb{R}^D$
  for t = 1, 2, ... do
      Retrieve $\mathbf{x}_t$
      $\hat{y} = \mathsf{sign}(\mathbf{w}_t \cdot \mathbf{x}_t)$
      Retrieve $y$
      if $\hat{y} \neq y$ then
          $\mathbf{w}_{t+1} = \mathbf{w}_t + y_t\mathbf{x}_t$
      end if
  end for
```

A book titled "Perceptron" [54] published by Minsky & Papert shows that simple linear models such as the perceptron cannot perfectly classify data that is not linearly separable and famously the XOR function,

$$\mathsf{XOR}(x_1, x_2) = -\mathsf{sign}(x_1 \cdot x_2) \quad \text{where, } x_1, x_2 \in \{-1, +1\} \tag{2.2}$$

Fortunately by stacking layers of parallel perceptrons we can construct a parameterisation that precisely captures the XOR function, see [4, Ch. 6.1]. However, for arbitrary functions we will first need to introduce the artificial neuron. The artificial neuron generalises the perception by replacing the sign function with some arbitrary activation function g, also called the *transfer function*. Mathematically the artificial neuron computes the following function,

$$y = g\left(\sum_{i=1}^{D}(w_i \cdot x_i) + b\right) \tag{2.3}$$

where D is the dimension of the input feature vector $\mathbf{x}$. We can stack multiple artificial neurons in parallel over the same inputs. Given that each artificial neuron has its own set of learnable weights and biases, we may expect each neuron to compute different output values, see Figure 2.2.

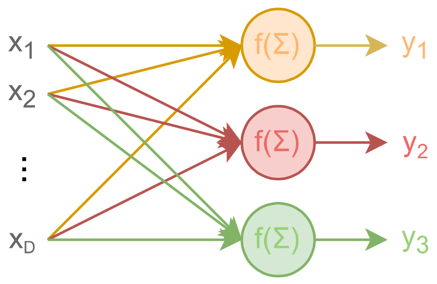

Figure 2.2: 3 stacked artificial neurons forming a layer of a neural network

When we stack artificial neurons in this way we form what is known as a *linear* or a *fully connected layer*. Importantly, the operation applied by a single linear layer can be conveniently expressed in terms of matrix and vector operations,

$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.4}$$

where,

- $\mathbf{y}$ denotes the output (column) vector with dimension N, where N corresponds to the number of stacked artificial neurons.

- g denotes the activation function typically applied element-wise to the pre-activation vector $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$.
- $\mathbf{W}$ denotes the $N \times D$ matrix of weights.
- $\mathbf{x}$ denotes the D-dimensional vector of inputs or activations from the previous layer.
- $\mathbf{b}$ denotes the D-dimensional vector of biases.

Linear layers of this form followed by non-linear activation functions are referred to as *hidden layers* in the literature, and the neurons that make up the hidden layers are called *hidden units*. Which activation function to use entirely depends on the task at hand. The *Sigmoid* and *Softmax* activation functions functions are typically used for classification tasks due to their convenient properties. Other activation functions include the, *rectified linear unit* (ReLU) function and the *hyperbolic tangent function* (*tanh*), which are commonly used for internal layers of the network. Table 2.1 lists some of the most common activation functions we expect to encounter and their mathematical definitions.

Activation Function	Equation	Properties
Identity	$g(x) = x$	This reduces a single linear layer to a linear regression model.
Sign	$g(x) = \mathsf{sign}(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$	The perceptron model.
Binary	$g(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$	$\{0, 1\}$ classifier
Sigmoid	$g(x) = \frac{1}{1+e^{-x}}$	"Soft" differentiable version of the binary classifier.
Softmax	$g(x) = \frac{1+e^{x}}{\sum_{i=1}^{D} 1+e^{x}i}$	Multiclass sigmoid applied at the layer level not element-wise.
ReLU	$g(x) = \mathsf{max}\{0, x\}$	Acts like the identity unless $x < 0$.
Tanh	$g(x) = \frac{2}{1+e^{-2x}} - 1$	Similar to Sigmoid but with range [-1, 1].

Table 2.1: Common activation functions

2.1.2 Multi-layer Perceptrons

Multi-layer perceptrons (MLPs) [4, 51] also called feed-forward neural networks are networks composed of more than one layer of artificial neurons. Layers of artificial neurons are organised in a sequential manner so that the output of one layer is fed as the input into the next layer. Figure 2.3 is provided as a visual example.

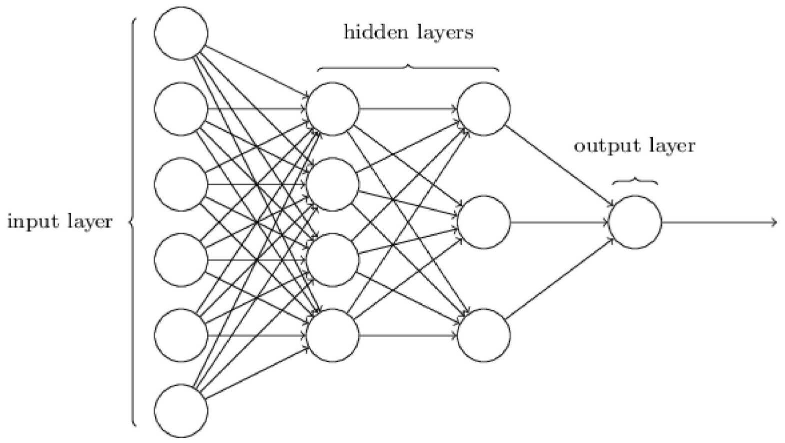

Figure 2.3: Multi-layer perceptron with 2 hidden layers

The network given in figure 2.3 can be viewed as a composition of functions $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$, where each of the functions $f^{(1)}, f^{(2)}, f^{(3)}$, represent the computation at each layer. We can think of each layer as computing a non-linear representation of the features of the previous layer; a good learning algorithm must decide how to best use and learn the internal representations at each hidden layer in order to best match the outputs to the target outputs [4, p. 164]. The *depth* of a network refers to the number of hidden layers used; this is essentially where the name deep learning comes from. The *width* of a network refers to the number of hidden units or artificial neurons in each layer of the network. For example, in figure 2.3 the network consists of 2 hidden layers, the first of which has 4 hidden units and the second has 3.

Input features (measurements, pixel value etc.) are fed to the input layer and subsequently processed by the hidden layers until the output value is computed. While the definition of an MLP is not necessarily very strict, we will assume that all the hidden layers of an MLP are fully connected. With this configuration in mind, the full computation performed by an MLP can be expressed by the *forward equations*:

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l \tag{2.5}$$

$$\mathbf{a}^l = g^l(\mathbf{z}^l) \tag{2.6}$$

where,

- $\mathbf{z}^l$ denotes the *pre-activations* of the l^{th} layer.
- g^l denotes the activation function used at the l^{th} layer.
- $\mathbf{W}^l$ is the weight matrix for the l^{th} layer.
- $\mathbf{b}^l$ is the vector of biases for the l^{th} layer.
- $\mathbf{a}^{l-1}$ and $\mathbf{a}^l$ denote the *activations* of the $l-1^{\text{th}}$ and l^{th} layer respectively.

By convention we set the first pre-activation and activation to the input vector:

$$\mathbf{z}^1 = \mathbf{a}^1 = \mathbf{x} \tag{2.7}$$

An MLP with L layers has $L-1$ pairs of weights and biases:

$$\boldsymbol{\theta} = \{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L, \mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\} \tag{2.8}$$

By repeatedly applying the forward equations we can make a prediction $\hat{y}$ given some input feature vector $\mathbf{x}$. This is called the *forward pass.*

The key benefit of stacking hidden layers is that we can now capture non-linear dependencies and approximate non-linear functions. In other words, MLPs offer us with a much more expressive class of models than traditional linear models like the perceptron. An important result by Hornik *et al.* showed that MLPs can approximate any function on a closed interval to an arbitrary degree of accuracy when the number of hidden units in a single layer is unconstrained [55]. This result is known as the *universal function approximation* result, which claims in theory, that MLPs are *universal function approximators*; they can approximate any target function f^*. Although in general, finding a parameterisation that closely approximates the function we care about, f^*, is not easy. In machine learning the problem of finding such a parameterisation is known as *training*.

2.1.3 Optimisation and Backpropagation

Training neural networks is typical done via gradient-based optimisation, a common technique used in many more traditional machine learning algorithms as well. In high school, when we are asked to find the minima of a function, our first instinct is to compute its derivative and set this to zero. Modern machine learning algorithms are based on exactly the same principle, although in practice for higher dimensional models solving the gradient equation is infeasible, and instead we rely on general purpose methods such as *gradient descent*. Gradient descent works by iteratively updating the parameters of our model $\boldsymbol{\theta}$ in the direction that minimises some differentiable objective function $J(\boldsymbol{\theta})$ as follows,

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}) \tag{2.9}$$

where η is a (scalar) learning rate parameter, typically set to a small positive value that scales the gradient update steps. If the objective function is convex, then provided η is small enough, we are guaranteed to converge to the global minima, no matter where we start in the parameter space. Unfortunately, when it comes to training neural networks we are rarely afforded such guarantees. The problem is typically is non-convex [56]. This fact is easy to show, since there exist many parameterisations of the same MLP that compute the same function (simply permute the hidden units). Nevertheless, we still apply convex optimisation techniques similar to gradient descent for training neural networks, but several additional tricks are typically required.

For gradient descent to work we require the objective function $J(\boldsymbol{\theta})$ to be continuous and differentiable with respect to the parameters $\boldsymbol{\theta}$. For MLPs this means any of the activation functions we use must be differentiable with respect to their inputs. In classification tasks our aim is to minimise the number of miss-classified examples of some given dataset $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{n}$. For most useful machine learning models including MLPs, finding the set of parameters that minimises the number of missclassified examples is intractable. Instead we specify a differentiable objective function called the *loss function*, that when minimised should improve the performance of our model on the task at hand.

The choice of loss function tends to be critical to the final performance of the network. The loss function needs to reflect the given task in such a way, that by minimising the loss function we expect perform well at the given task. In most cases we employ the principle of maximum likelihood: in a probabilistic sense MLPs typically model some conditional distribution $p(y \,|\, \mathbf{x}, \boldsymbol{\theta})$, and we wish to recover the parameters $\boldsymbol{\theta}$ that maximise the likelihood of the data under this distribution. This should in theory provide the best explanation of the data at hand and best capture the conditional distribution.

In this thesis we will use *deep Q-networks* (DQNs) [19, 20] which approximate the Q-values of state-action pairs; this task can be viewed as a form of regression. For regression tasks our output features typically approximate the mean of a Gaussian distribution. In this case, minimising the *mean-squared error* (MSE) is equivalent to maximising the likelihood of the data under the conditional distribution $p(y|\mathbf{x}, \theta)$ [4, p. 176]. So for DQNs we typically use the MSE loss function as the objective function we seek to minimise,

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.10}$$

In this thesis we also cover the cross-entropy method, a simple but effective method for solving relatively simple RL problems. The details are left for later, but effectively the cross-entropy method is a form of classification, where the underlying distribution from which the examples are drawn from changes slightly each batch iteration. Artificial neural networks (ANNs) used for the cross-entropy method implement the $\mathsf{softmax}$ function in their linear output layer, see Table 2.1. This is because the softmax function is differentiable with respect to its inputs and it gives us a categorical distribution on the output features, which we can sample from. In this scenario minimising the cross entropy loss is equivalent to maximising the likelihood of the conditional distribution $p(y|\mathbf{x}, \theta)$ [4, p. 178]. The cross entropy loss for probability vectors $\mathbf{y}$ and $\hat{\mathbf{y}}$ is given below,

$$\mathcal{L}_{\text{cross entropy}} = -\frac{1}{n}\sum_{i=1}^{n} \mathbf{y} \cdot \log(\hat{\mathbf{y}}) \tag{2.11}$$

Now that we are equipped with some useful differentiable loss functions we can start to think about computing gradients. The backpropagation (BP) algorithm [12] provides us with a means for computing the gradients of the parameters of a neural network $\boldsymbol{\theta}$ with respect some loss function $\mathcal{L}$. Once we have computed these gradients we can begin to optimise the model by applying gradient based optimisation techniques, like gradient descent. Essentially, BP is a straightforward application of the chain rule of differentiation and it forms the basis for almost all gradient based learning algorithms for neural networks. Given some set of input-output targets or dataset $\mathcal{D}$, BP works in four stages stages:

1. **Forward pass:** feed the inputs $\mathbf{x}$ through the MLP, by applying the forward equations (Equations 2.5 and 2.6) to obtain the corresponding output value $\hat{y}$.

2. **Compute the loss:** under the criteria specified by the loss function $\mathcal{L}$, compute the scalar loss by comparing the outputs $\hat{y}$ computed in the forward pass to the output targets y provided in the dataset.

3. **Backward pass:** back-propagate the loss though the network by applying the chain rule of differentiation to compute the parameter gradients.

4. **Parameter update:** update the parameters of the network $\boldsymbol{\theta}$ in the direction that minimises the loss function $\mathcal{L}$ according to some optimisation procedure.

More formally, Algorithm 2 outlines the general procedure for a training on a single data point. We refer the reader to Appendix 2 for a formal derivation of the error backpropagation procedure outlined by Algorithm 2. It should be clear that the forward pass and parameter update steps outlined in Algorithm 2 are immediately derived from the forward equations (Equations 2.5 and 2.6) and the gradient descent update step (Equation 2.9) respectively. We also note that the procedure outlined by Algorithm 2 can easily be extended to batch data by replacing the matrix multiplications with appropriate tensor operations.

Algorithm 2 Backpropagation

```
Input: inputs $\mathbf{x}$, targets $\mathbf{y}$, weights $\{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L\}$ and biases $\{\mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\}$.
Output updated weights $\{\mathbf{W}^2, \mathbf{W}^3, ..., \mathbf{W}^L\}$ and biases $\{\mathbf{b}^2, \mathbf{b}^3, ..., \mathbf{b}^L\}$.
  $\mathbf{a}^1, \mathbf{z}^1 \leftarrow \mathbf{x}^1$
  for $l \leftarrow 2$ ; $l < L + 1$ ; $l \leftarrow l + 1$ do                 ▷ Forward pass
      $\mathbf{z}^l \leftarrow \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$
      $\mathbf{a}^l \leftarrow g^l(\mathbf{z}^l)$
  end for
  $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^L} \leftarrow \nabla_{\mathbf{z}^L} \mathcal{L}(\mathbf{y}, \mathbf{a^L})$                              ▷ Compute the loss
  for $l \leftarrow L$ ; $l > 2$ ; $l \leftarrow l - 1$ do                    ▷ Backward pass
      $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{l-1}} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathbf{z}^l} \mathbf{W}^l \frac{\partial g^l}{\partial \mathbf{z}^l}$
  end for
  for $l \leftarrow 2$ ; $l < L + 1$ ; $l \leftarrow l + 1$ do                 ▷ Update the parameters
      $\Delta \mathbf{W}^l \leftarrow \eta \left[\mathbf{a}^{l-1} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^l}\right]^T$
      $\mathbf{W}^l \leftarrow \mathbf{W}^l + \Delta \mathbf{W}^l$
      $\Delta \mathbf{b}^l \leftarrow \eta \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}^l}\right]^T$
      $\mathbf{b}^l \leftarrow \mathbf{b}^l + \Delta \mathbf{b}^l$
  end for
```

An important property of BP is that it can compute gradients locally, each layer only needs l to know about gradients of the next layer $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{l+1}}$ and the activations at the previous layer $\mathbf{a}^{l-1}$. This property leads to efficient implementations of BP and provides some justification of its biological plausibility [51, p. 572], although this is a widely debated interpretation [29, 30].

Typically in modern deep learning we are given access to large datasets, and so computing gradients over the entire dataset $\mathcal{D}$ is very costly. Instead, we typically employ a batch gradient descent algorithm called *stochastic gradient descent* (SGD). With SGD, we sample a mini-batch of input-output pairs from $\mathcal{D}$ and compute the gradients using this much smaller amount of data. Because of the randomness associated with sampling mini-batches, the parameter updates can become noisy which may cause divergence during training, to deal with this issue we can either increase the mini-batch size or adjust the learning rate parameter η accordingly.

In the deep learning literature there exists a vast array of mini-batch gradient optimisers, all built upon the core SGD algorithm. SGD is very sensitive to the learning rate parameter η; if η is set too big we will likely overshoot the minima and diverge, if η is set too small we will converge very slowly to the minima. Clever scheduling and annealing of the learning rate can help but this just gives us another thing to tune.

The *Adam* optimiser [43] is an optimisation scheme than has found a lot of success in RL and DL as a whole. The Adam optimiser is fairly sophisticated algorithm that is less sensitive to the user-specified learning rate η and is generally treated as a black box optimiser. Although, we must note that in some settings Adam still appears sensitive to the choice of learning rate, however it is still preferred over SGD since it converges much quicker in practice.

The Adam update procedure is based on two key ideas:

- Adapting the learning rate for each parameter based on the exponentially weighted moving average of its previous gradient norms.
- Using the exponentially weighted average of the gradients momentum for each parameter.

Algorithm 3 below outlines Adam update for a single parameter w.

While the Adam optimiser has 3 hyper-parameters η, β_1 and β_2, the betas are typically left as their default values $\beta_1 = 0.9$ and $\beta_2 = 0.999$. In some of our work we will use Adam, as it simply converges much quicker in many challenging RL scenarios, although we must note that Adam is optimised for backpropagation and not for alternative gradient based learning algorithms such as predictive coding. So in general we will try use SGD

Algorithm 3 Adam Optimiser

```
Input: $\eta$ (learning rate), $\beta_1$, $\beta_2$, $w_0$ (parameter), $L(w)$ (loss function)
Initialize: $m_0 = 0$, $v_0 = 0$
  for t = 1, 2, ... do
    $g_t \leftarrow \nabla_w \mathcal{L}(w_{t-1})$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
    $w_t \leftarrow w_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$
  end for
```

where possible in the interest of fairness and explainability. Furthermore, we note that the development of a more effective gradient based optimiser built on the predictive coding algorithm is an important area of orthogonal research.

2.1.4 Convolutional Neural Networks

We have already alluded to the idea that the hidden units of neural networks learn non-linear representations of the features of the previous layer; this is what makes MLPs more expressive than linear models. However, MLPs have their fair share of shortcomings in practice and commonly suffer from *overfitting* to the dataset. Overfitting in machine learning refers to the phenomena whereby a statistical model fits perfectly or very closely to the training data and achieves poor generalisation on unseen data. A common indicator of overfitting is when we observe excellent accuracy on the training dataset but poor accuracy on the test dataset. The idea of overfitting can be traced back to the famous *Occam's razor* principle, which states: when it comes to comparing theories or explanations, the simplest one, is preferred to the more complex one.

Convolutional neural networks (CNNs) [8] inspired by principles like Occam's razor and signal processing in the visual cortex have achieved much better generalisation capabilities than standard MLPs in tasks such as image classification, signal processing and representation learning. CNNs are well suited to 1d signals such as audio, 2d signals such as images, and even 3d signals such as MRI scans and other health data [51, p. 565]. CNNs use convolution filters with learnable weights to process input signals. For 2d signals this leads to *weight sharing* across the input image and as a result the hidden units of a convolutional layer have local 2-dimensional *receptive fields* [51, p. 565] and often *sparse activations*. The intuitive property of this spatial parameter tying is that learned convolution filters can detect the same feature or pattern anywhere in the input image, without having to independently learn the same weights as we would if we used a standard MLP [51, p. 565]. This leads to *translation invariance*, meaning we can detect a specific pattern even if it is translated in the image. Another added benefit of CNNs is that we can apply them to variable size input, whereas the input layer of an MLP must have a rigid structure.

Let's consider some 2d input signal I, for example a grey-scale image consisting of raw pixel data. The mathematical operation applied by a convolution filter K at a single point in the image (i, j) is typically denoted by an asterisk $*$ and is expressed as follows,

$$S(i,j) = (K * I)(i,j) = \sum_{m=1}^{W} \sum_{n=1}^{H} I\left(i + m - \left\lfloor \frac{W}{2} \right\rfloor, j + n - \left\lfloor \frac{H}{2} \right\rfloor\right) \cdot K(m,n) \qquad (2.12)$$

where the convolution filter K has width W and height H. Typically (W, H) are equal and take values in $\{(3,3),(5,5),(7,7),...\}$. Figure 2.4 presents a visual interpretation of the convolution operation.

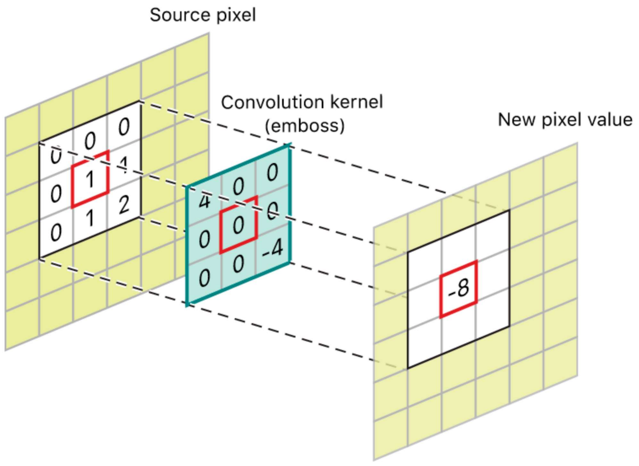

Figure 2.4: The convolution operation (Equation 2.12) applied by a convolution filter to a specific point in the input image

Typically, convolution filters are applied across the entire input in a sliding window fashion. If we want to apply the convolution filter K to every pixel in the image we can systematically shift the filter by one pixel (stride of 1) across the entire image and what we get back is called a *feature map*. In this scenario we will likely need to pad the image with zeros at its boundaries. We can also change the *stride* of the convolutional layer and skip every other pixel as opposed to applying it to every pixel, for more details about stride and zero-padding please refer to [57].

A convolutional layer in a neural network typically consists of many learnable convolutional filters applied in parallel, with each outputting their own feature map. It is also typical to use a *pooling layer* after a convolutional layer, since the same feature may have been picked up in two neighbouring pixels, and this is essentially redundant information. Pooling works by replacing each location in the output feature map with a summary statistic of nearby values [4, p. 330]; essentially we reduce the size of the feature maps by removing redundant or repeated feature detections in small local regions of the feature map, see figure 2.5.

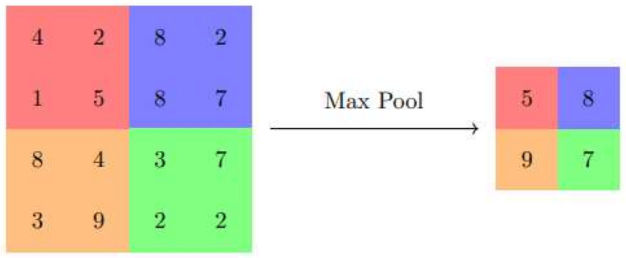

Figure 2.5: Max pooling operation. Pool size is 2x2 and stride is (2, 2).

Typically after several convolutional and pooling layers the output feature maps are flattened and passed through an MLP classifier or regressor. The feature extraction and processing performed by the convolutional and pooling layers is similar to internal representations learned in the hidden layers of a standard MLP, although exploitation of the local geometry of the input is what helps CNNs achieve better generalisation capabilities. Figure 2.6 illustrates the typical architecture of a CNN designed to classify images.

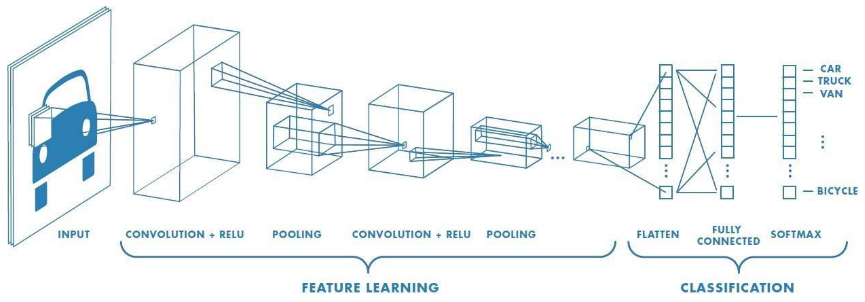

Figure 2.6: Typical architecture of a CNN images classifier.

Since the inception of CNNs, much of the recent research has become more of an engineering and mathematical effort than a neuroscience one. Important challenges that have been overcome include: successfully applying backpropagation to CNNs [1, 8], utilising deeper CNN architectures while avoiding the vanishing gradient problem [58], and adopting tricks such as dropout [13] and batch normalisation layers [59] to achieve better generalisation. Key successes from the past decade include AlexNet [1] for image classification and achieving super-human performance on Atari 2600 games by learning from raw pixel data [19, 20]. The later work will be used as the basis for much of the research in this thesis and its importance will become apparent.

2.2 Predictive Coding and Energy-Based Models

Predictive coding (PC) is a promising theory of learning in the brain first used as a means of explaining neural processing in the retina and visual cortex [33, 34]. More recently PC has evolved into a general purpose Hebbian learning algorithm that can be applied to networks of arbitrary topology [28, 40, 41]. Driven by local updates, PC has strong mathematical foundations and links to theories of Bayesian inference in the brain [35, 60]. In addition, PC posits as a more biologically plausible algorithm compared to BP and could be the key to achieving greater cognitive flexibility for systems aiming to capture human intelligence [42].

The idea behind PC is that brain function can be captured by inference and learning in some deep generative model. The architecture of such a model can be expressed in a