Department of Mathematics
and Computer Science

University of Southern Denmark

Masters Thesis in Computer Science

AttentionSplit:

A Hybrid Neural Network Architecture for Long- and Short Term Retention

Author
Frederik Gram Kortegaard
frkor19@student.sdu.dk

Author
Lasse Usbeck Andresen
lasan19@student.sdu.dk

Supervisor
Melih Kandemir
kandemir@imada.sdu.dk

July 7, 2024

Contents

1 Abstract

Machine learning, and in particular the area of deep-learning has shown great progress in many key applications, and as such, improvements to training regimes and model architectures are important fields of research within the literature. We present a new Adam variant based on the works of [40] that we discovered in the process of designing and testing and a new Artificial Neural Network layer architecture, which combines the ideas of recurrence and attention. Our Adam variant achieves improvements in maximal accuracy over it's parent algorithms, while the AttentionSplit module shows indications of improvements over other attention and recurrence modules. We conclude that area of research suggested by [40] is indeed very important for improving the performance of modern deep-learning models and conclude that while AttentionSplit is not fully matured as a concept, even this simple combination has great potential.

2 Introduction

The field of Machine-Learning has shown great results in many practical applications such as Computer Vision tasks, machine translation, data classification and control tasks. One area of research in the literature that has shown great promise over the last decade concerns models specialized in temporal data analysis, which deals with topics such as textual analysis, sequence predictions and classifications. Two particular paradigms within architectural design of models for these problem setting have gained precedence over the years, namely attention-based architectures - in particular the Transformers of [33] and recurrence-based architectures - which is exemplified by the *LSTM* [44] architecture.

In this paper will first introduce a new neural network architecture which we have dubbed *AttentionSplit*, which attempts to combine *attention-* and recurrence techniques. In connection with this, we showcase results which we have gathered using a variety of temporal data analysis tasks, which indicate the possibility of an increase in maximal accuracy on said tasks - with some important contradictory examples.

Furthermore, we propose a potential optimization of an *Adam* [31] variant optimizer based on concepts from *AdamP* [40], which we have named *OrthAdam*. As with *AttentionSplit*, we showcase results, which indicate further increases in potential maximum accuracy gains on multiple computer-vision based tasks. Based on these results, we find it probable that further value could be gained from more extensive research within this area, in conjunction with the sentiment of the *AdamP* [40] authors.

3 Prerequisites

3.1 Terminology

To serve as a basis for the terminology used in this work, we have chosen a pair of texts to serve as our baseline. We have chosen the follow the sets of symbols used for Calculus from the *Calculus 3* [22] book, and notation for Deep-Learning terminology where applicable from the *Optimizing Neural Networks with Kronecker-factored Approximate Curvature* [23] paper. For the sake of brevity, when we want to refer to the constructs of Artificial Neural Networks, we will refer to them simply as either Neural Networks, or with the acronym ANN. We will refer the layers of ANN's simply as it's layers or hidden layers, when not referring to the input or output layers explicitly. We will use standard notation for matrices and vectors, such that a matrix will be uppercase single bold letters e.g.$\mathbf{A}$, which refers to a matrix, while $\mathbf{a}$ refers to a vector. When we refer to element-wise operations, we shall explicitly state them to differentiate between these and the standard matrix and vector operations from Linear Algebra, although for some algorithms, we will abuse notation of cardinality from set theory applied to our data containers, for convenience sake.

When we take a look at the problems of Gradient Optimization theory, we will use the standard notation from Calculus and Probability theory, such as when giving a brief overview of back-propagation, we will denote the partial derivatives through the chain-rule

$$\frac{\partial w}{\partial t_j} = \frac{\partial w}{\partial x_1}\frac{\partial x_1}{\partial t_j} + \frac{\partial w}{\partial x_2}\frac{\partial x_2}{\partial t_j} + ... + \frac{\partial w}{\partial x_m}\frac{\partial x_m}{\partial t_j}$$

given

$$w = f(x_1, x_2, ..., x_m)$$

for m independent variables, with

$$x_i = x_i(t_1, t_2, ..., t_n)$$

for

$$i \in \{1, 2, ..., m\}$$

Likewise, we shall use

$$\mathbf{s}_i = \mathbf{W}_i^\top \mathbf{a}_i + \mathbf{b}_i$$

to indicate the linear output of a model at layer i and

$$\mathbf{a}_{i+1} = \phi(\mathbf{s}_i)$$

to describe the non-linear activation of the i'th layer of a model, where ϕ is any given non-linear activation function. Given the parameters of a model, we shall use the common notation of θ to symbolise the "flattened" and concatenated parameters. When describing the final output of the collective activation of the model, we simply use $f(x, \theta)$ to describe the model's prediction given input x and parameters θ. To describe the error or loss that is measured between a model prediction versus a given label or target, y is denoted as

$$L(y, f(x, \theta))$$

The exact measure of the loss function changes depending on the task, but for most simple cases, the mean-squared error is used. Likewise, the target y is defined on a per-task basis.

3.2 Theory

For the theory section, we decided to include information that would not be considered necessarily advanced knowledge, such as theory from multivariate calculus. This is because such courses are not mandatory for our level of education, and as such, we feel compelled to explain to readers these concepts. This is because this work is intended to be accessible to other computer science graduates with similar backgrounds as ourselves. For brevity, we will omit most proofs of multivariate calculus and refer to those found in [22] as to focus on necessary definitions.

3.2.1 Training ANN's

Before discussing Recurrent Neural Networks, it would be good to ensure an understanding of how training ANN's work. ANN's are trained to minimize a given loss function, and works via. mathematical optimization, most commonly by way of gradient descent. In order to understand gradient descent in the context of ANN's, we must first understand how it relates to the concept of partial derivatives, the chain rule, directional derivatives and the gradient.

First we consider a partial derivative. Partial derivatives come into play, when we deal with functions of multiple independent variables. Consider a function of two independent variables $z = f(x, y)$. The partial derivative is then defined as the slope of the tangent line parallel to one of the given axis e.g. either x or y. Thus, in this case, we will have two possible different partial derivatives, both defined from $z = f(x, y)$. Therefore, from [22] sections 4.12 and 4.13, we get the definition:

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h, y) - f(x, y)}{h}$$

This describes the rate of change in the direction parallel to the x-axis for $f(x, y)$. Thus we see that we can define a partial derivative for a function that consists of arbitrarily many independent variables, by letting the other independent variables excluding the one under consideration, remain constant. For our purposes, we need not consider higher-order partial derivatives.

We now want to consider the slope of a tangent line along any direction, and not only those parallel to one of the axis defined by a function. Following from [22], we define first an angle θ and a distance of h to travel, which will in turn define the direction we wish to travel. Given $f(x, y)$, and a set of values such that $x = a$ and $y = b$, then a point belonging to the domain of $f(x, y)$ will be $(a, b, f(a, b))$. A second point can then be found belonging also to the domain, in the direction of an unit vector $u = cos(\theta)\mathbf{i} + sin(\theta)\mathbf{j}$, which we will give the magnitude of the chosen distance h. It follows then that $(a + hcos(\theta), b + hcos(\theta), f(a + hcos(\theta), b + hcos(\theta))$ is the second point in the direction and distance desired. From the definition of the secant line and limit, we can then use these points to define the tangent line through a point that is in a given direction $\mathbf{u}$:

$$D_{\mathbf{u}} f(a, b) = \lim_{h \to 0} \frac{f(a + hcos(\theta), b + hcos(\theta) - f(a, b)}{h}$$

Thus, we can define the rate of change in any direction for a function differentiable in a point $(a, b, f(a, b))$. Alternatively, instead of using $\mathbf{i}$ and $\mathbf{j}$ to define direction and then finding the limit as h goes to 0, we can use the partial derivatives - the proof of which can be found in [22] as theorem 4.12 - as such:

$$D_{\mathbf{u}} f(x, y) = f_x(x, y)cos(\theta) + f_y(x, y)sin(\theta)$$

In fact, we will see that when we generalize to functions of more than two variables, the partial derivatives will function as the way to compute directional derivatives. In these cases, we will define our direction using $\mathbf{u} = cos(\alpha)\mathbf{i} + cos(\beta)\mathbf{j} + cos(\gamma)\mathbf{k}$. From this, we see that this unit vector has components all defined using cosine. As states by [22], these are called directional cosines of $\mathbf{u}$.
In general, for a function in $\mathbb{R}^n$, we can use vector $\mathbf{u}$ to define a direction, and then use the definition of a limit to calculate the rate of change in that direction, for a function which takes some vector $\mathbf{r}_0$ as the initial point:

$$D_{\mathbf{u}} f(\mathbf{r}_0) = \lim_{h \to 0} \frac{f(\mathbf{r}_0 + h\mathbf{u}) - f(\mathbf{r}_0)}{h}$$

With this working definition of n-dimensional directional derivatives, we will work on defining the direction of greatest change for the directional derivative.

We now want to look at the directional derivative that has the direction of the greatest amount of change. This directional derivative is what is referred to the gradient. The gradient is defined to be a linear combination of partial derivatives for the function, with the unit vector for the axes to serve as the basis. For a function with three independent variables:

$$\nabla f(x, y, z) = f_x(x, y, z)\mathbf{i} + f_y(x, y, z)\mathbf{j} + f_z(x, y, z)\mathbf{k}$$

Thus we see the gradient is well defined for $\mathbb{R}^n$, if the partial derivatives exists for $f(\mathbf{x})$. Some important properties of the gradient include:

- if $\nabla f(\mathbf{x}) = \mathbf{0}$, then for all $\mathbf{u}, D_{\mathbf{u}} f(\mathbf{x}) = \mathbf{0}$
- if $\nabla f(\mathbf{x}) \neq \mathbf{0}$, then for all $\mathbf{u}, D_{\mathbf{u}} f(\mathbf{x}) <= \nabla f(\mathbf{x})$

This can be summarized as saying, that if the gradient is equal to zero, the derivative in any direction other than the gradient will also be zero. If the gradient is not zero, it will be a directional derivative, whose values are maximal. This leads to the following two important realizations: continuously "following" the gradient, will lead to a local maxima, while going in the opposite direction will in turn go to a local minima. This concept is the basis of the gradient descent algorithm.
But before discussing gradient descent in these cases, we will need to show how to calculate the gradients for an ANN, for which we will need a definition of multivariate chain-rule.

For multivariate functions, we are still able to define a useful definition of a chain-rule. Starting out with a simple scenario, as in [22], Given a function defined in two independent variables x and y, $z = f(x, y)$, these independent variables could be functions in and of themselves. Assume both x and y is defined for the independent variable t, then $z = f(x(t), y(t))$ and we can differentiate with z with respect to t:

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt}$$

We calculate the derivatives of the inner functions $x(t), y(t)$ with respect to t, and calculate the partial derivatives of the outer function z with respect to it's independent variables to obtain a final derivative.

Next we will focus on the case where both inner functions of z, x and y are both multivariate functions, with independent variables u, v, such that $x(u, v)$ and $y(u, v)$, then $z = f(x(u, v), y(u, v))$. In this case, we will get two partial derivatives, $\frac{\partial z}{\partial u}$ and $\frac{\partial z}{\partial v}$, as opposed to the singular derivative $\frac{dz}{dt}$ from the previous example. This is because, in the first case, z was only dependent on t through transitivity, and in this case, where x and y are themselves dependent variables of two independent variables, the outer-most function is then dependent on both the inner-most independent variables u and v. The partial derivatives of z are defined as:

$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial u}$$

$$\frac{\partial z}{\partial v} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial v} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial v}$$

We see in this case, the partial derivatives are made up of different partial derivatives. The reasoning follows along the same line as the partial derivatives for z. Since both x and y are now dependent on two independent variables u, v, there is no longer a singular derivative for x and y, but instead two partial derivatives with respect to u and v.

We will now re-state the generalized chain rule, as we did in the prerequisites:

$$\frac{\partial w}{\partial t_j} = \frac{\partial w}{\partial x_1}\frac{\partial x_1}{\partial t_j} + \frac{\partial w}{\partial x_2}\frac{\partial x_2}{\partial t_j} + ... + \frac{\partial w}{\partial x_m}\frac{\partial x_m}{\partial t_j}$$

given

$$w = f(x_1, x_2, ..., x_m)$$

for m independent variables, with

$$x_i = x_i(t_1, t_2, ..., t_n)$$

for

$$i \in \{1, 2, ..., m\}$$

We then see, that this definition gives us the outer-most function w, depending on the set of inner functions x_i, where each x_i is in turn defined by t_n independent variables. Therefore, we get as many partial derivatives, as there are combinations of the inner-functions and their independent variables.

Now we take a look at the structure of a standard feed-forward ANN, to see how these concepts will combine. ANN's are build up from multiple "layers", where each layer i, for $i = \{1, 2, 3, ..., n\}$, is a matrix component $\mathbf{W}_i$ with a vector component $\mathbf{b}_i$ and an associated non-linear activation function ϕ_i. To describe the general structuring of each of these layers, we will consider a scenario consisting of some integer number n inputs and some integer number m outputs. A model with i layers will consist of an input layer, with h rows and n columns in it's associated matrix and a vector with h components. Thus the first activation can be described by:

$$\mathbf{s}_0 = \mathbf{W}_0^\top \mathbf{a}_0 + \mathbf{b}_0$$

$$\mathbf{a}_1 = \phi_0(\mathbf{s}_0)$$

$\mathbf{s}_0$ is a linear mapping $S : \mathbb{R}^n \to \mathbb{R}^h$. To avoid all layers of the model ultimately becoming a linear mapping $S : \mathbb{R}^n \to \mathbb{R}^m$, we use a non-linear activation ϕ, to introduce non-linearity between layers. This structure is followed for the remaining layers, typically excluding the non-linear activation function in the final layer, such that for the i'th layer, we get:

$$\mathbf{s}_i = \mathbf{W}_i^\top \mathbf{a}_i + \mathbf{b}_i$$

$$\mathbf{a}_{i+1} = \phi(\mathbf{s}_i)$$

Given this description of a general ANN structure, consider a problem statement as follows: Given a specific input $\mathbf{a}_0$ of n components, choose between m outputs. The network produces a vector $\mathbf{x}$ with m components, from which the maximal valued component is typically chosen. Consider then a given $\mathbf{y}$ with m components also that contains the "true" or target values associated with the given $\mathbf{a}_0$. We can then differentiate between the network prediction and the true values in $\mathbf{y}$ to define an error or loss function:

$$L(\mathbf{x}, \mathbf{y})$$

Typically, to indicate that $\mathbf{x}$ is a composite function of the network, we write

$$L(f(\theta, \mathbf{a}_0), \mathbf{y})$$

where θ is describing the current network parameters, and that L produces a singular numerical value. Assume this function accurately represents the difference between the "true" m values associated with $\mathbf{a}_0$, then we want to find parameters θ^* among the set of possible parameters for the model, such that L is minimized.

We know from the earlier properties of given of the gradient that adjusting the independent variables in the direction of the gradient will maximize a function, while following the opposite direction will minimize a function. Therefrom it follows that adjusting each parameter of the model in an iterative fashion according to the gradient of each parameter will eventually lead to maximizing or minimizing a loss function.

Starting at the output layer, there are m components to $f(\theta, \mathbf{a}_0)$, so it follows that there are m partial derivatives, i.e. one for each parameter:

$$\frac{\partial L}{\partial \mathbf{x}_m}$$

These partial derivatives each define the gradient for a particular parameter in the output layer, as the parameters of the output are not dependent on each other. Thus, a particular partial derivative $\frac{\partial L}{\partial \mathbf{x}_m}$ will be a gradient to the corresponding matrix vector pair.

Going back one step, each parameter matrix-vector pair is connected to h different activations in the next layer. Therefore, we must calculate all the h partial derivatives for each matrix-vector pair in the layer. Since the next layer, which is the output layer, is dependent on the loss function, we must use the chain-rule to calculate the partial derivatives with respect to this previous layer, since this previous layer is a composite function of the next layers in the back-propagation context. Using all these partial derivatives at the subsequent layer, we can calculate a gradient value for each of the matrix-vector parameters. This process is repeated through to the input layer and the process is summarized as Algorithm [1] from [23]. From this algorithm, numerous Gradient Descent algorithms can be defined to minimize the loss function.

Algorithm 1 An algorithm for computing the gradient of the loss $L(y, f(x, \theta))$ for a given (x, y). Note that we are assuming here for simplicity that the ϕ_i are defined as coordinate-wise functions.

```
input: $a_0 = x$; $\theta$ mapped to $(W_1, W_2, \ldots, W_\ell)$.

/* Forward pass */
for all $i$ from $1$ to $\ell$ do
    $s_i \leftarrow W_i \bar{a}_{i-1}$
    $a_i \leftarrow \phi_i(s_i)$
end for

/* Loss derivative computation */
$\mathcal{D}a_\ell \leftarrow \left. \frac{\partial L(y, z)}{\partial z} \right|_{z=a_\ell}$

/* Backwards pass */
for all $i$ from $\ell$ downto $1$ do
    $g_i \leftarrow \mathcal{D}a_i \odot \phi_i'(s_i)$
    $\mathcal{D}W_i \leftarrow g_i \bar{a}_{i-1}^\top$
    $\mathcal{D}a_{i-1} \leftarrow W_i^\top g_i$
end for

output: $\mathcal{D}\theta = [\mathrm{vec}(\mathcal{D}W_1)^\top \; \mathrm{vec}(\mathcal{D}W_2)^\top \ldots \mathrm{vec}(\mathcal{D}W_\ell)^\top]^\top$
```

Figure 1: Back-propagation algorithm

3.2.2 Gradient Based Optimization Theory

We have previously discussed the process of back-propagation to compute the gradient for each parameter in a ANN model. We also explained some necessary background for conceptualizing the gradient itself and properties of the gradient. given these tools, we have a general baseline, from which a multitude of important methods are based.

In most algorithms designed for training ANN models, the algorithms deal with networks on a per-parameter basis. By this we mean that once the gradient has been calculated, the parameters and corresponding gradients are treated as independent. Also, these calculations deal only with derivatives of the first-order and are often referred to in the literature as first-order methods. Methods that use higher-level derivatives - either calculated or approximated - also exists with the primary focus on methods studying the second-order derivatives. While such methods are theoretically very sound, they are often not chosen over first-order methods, because of practical limitations of computation.

3.2.2.1 Gradient Descent & Stochastic Gradient Descent One of the most well studied algorithms is based on one of the important properties of the gradient, namely the property of maximum direction:

$$\text{if } \nabla f(\mathbf{x}) \neq \mathbf{0}, \text{ then for all } \mathbf{u}, D_{\mathbf{u}} f(\mathbf{x}) <= \nabla f(\mathbf{x})$$

Therefore, going in the opposite direction of the gradient will instead minimize the function being considered. As can be seen in [1] - without the inclusion of the back-propagation algorithm -

Algorithm 1 Gradient descent

```
Require: $\theta$ = model parameters, $\alpha$ = learning rate, $t = 0$
Ensure: $\theta \in \mathbb{R}$, $\alpha \in (0, 1)$
  while $\theta \neq \theta^*$ do
    $\mathbf{g}_t = \nabla f(\theta_t(\mathbf{a}_0), \mathbf{y})$
    $\theta_{t+1} = \theta_t - \alpha \mathbf{g}_t$
    $t = t + 1$
  end while
```

Gradient Descent itself is relatively simple to compute on a per-parameter basis, with only a single hyper parameter inside our control. We simply repeat the computation of the loop until we reach convergence, i.e. we are at a local minimum.

So why do we not just stick with Gradient Descent? Logically, we know our method is guaranteed to reach a local minimum. The main reasons are two-fold:

- Computational limits of processing all data at once
- Reaching ”sharp” minima.

Gradient Descent in it’s core formulation requires us to evaluate the gradient on the entirety of the available data. Consider even a modest data set of 10000 samples. Now consider if each is a 32x32 image. We would then need to have all 10000 32x32 images in RAM at once and compute

the forward pass for it. Therefore, the first reason is of a mundane nature: it's not computationally practical.
The second reason is less obvious. In short, research has found that training with large sample sizes (in this case, all the samples) will lead to faster convergence, but also worse generalization capabilities [24]. In the context of this paper, our large batch would be the entire data set itself.

Consider instead the effect of computing the gradients based on a limited-size sample from the entire dataset. In short, this algorithm is more practical from a computational stand-point, as we can regulate the size of the batches of samples. In fact, studies have found that it is the ratio of the learning rate and batch-size [25], which impacts the minima achieved the most. We give a presentation of SGD as [2]. From this, it can be seen that the formulation itself follows closely from

Algorithm 2 Stochastic Gradient descent

```
Require: $\theta$ = model parameters, $\alpha$ = learning rate, $t = 0$, $batch - size$ = Number of samples per
  training loop.
Ensure: $\theta \in \mathbb{R}$, $\alpha \in (0,1)$, $|batch - size| <= |data|$
  while $\theta \neq \theta^*$ do
    $\mathbf{a}_t, \mathbf{y}_t = rand(batch - size, data)$
    $\mathbf{g}_t = \nabla f(\theta_t(\mathbf{a}_t), \mathbf{y}_t)$
    $\theta_{t+1} = \theta_t - \alpha \mathbf{g}_t$
    $t = t + 1$
  end while
```

Gradient Descent, but the implications of the stochastic variant on performance is vast.

3.2.2.2 Statistical Methods of Gradient Descent One popular topic in the literature is statistical methods of improving upon the baseline GD and SGD methods. Many of these methods are concerned with regularization terms.
One such popular method is called weight-decay. Weight decay is the process of reducing model complexity, where model complexity in this context is viewed as the sum of squared values of parameters $L2(model) = \Sigma_{i=0}^{n} \mathbf{W}_i^2$. In this view, a higher squared parameter value is indicative of greater model complexity. Therefore, a term reducing the absolute values of weights, will also reduce model complexity: In this case, we modify the gradient such that it also moves in the direction of the

Algorithm 3 SGD with weight-decay

```
Require: $\theta$ = model parameters, $\alpha$ = learning rate, $t = 0$, $batch - size$ = Number of samples per
  training loop, $\gamma$ = Weight-decay ratio
Ensure: $\theta \in \mathbb{R}$, $\alpha \in (0,1)$, $|batch - size| <= |data|$, $\gamma \in [0,1)$
  while $\theta \neq \theta^*$ do
    $\mathbf{a}_t, \mathbf{y}_t = rand(batch - size, data)$        ▷ Randomly sample data with batch-size datapoints
    $\mathbf{g}_t = \nabla f(\theta_t(\mathbf{a}_t), \mathbf{y}_t) + \theta_t \gamma$
    $\theta_{t+1} = \theta_t - \alpha \mathbf{g}_t$
    $t = t + 1$
  end while
```

current weight parameters. Since we are subtracting this gradient later on, this has the effect of also moving in the opposite direction of the current parameter values, e.g. making them smaller, from which the term weight-decay finds itself.

Another important method of modern SGD is momentum based methods, and in particular a type of momentum known as Nesterov momentum [30]. We present SGD with classical momentum as [4]

Algorithm 4 SGD with weight-decay & Classical Momentum

```
Require: $\theta$ = model parameters, $\alpha$ = learning rate, $t = 0$, $batch - size$ = Number of samples per
   training loop, $\gamma$ = Weight-decay ratio, $\mu$ = Momentum factor
Ensure: $\theta \in \mathbb{R}$, $\alpha \in (0, 1)$, $batch - size <= |data|$, $\gamma \in [0, 1), \mu \in [0, 1), \mathbf{v}_0 = \mathbf{0}$
  while $\theta \neq \theta^*$ do
    $\mathbf{a}_t, \mathbf{y}_t = rand(batch - size, data)$        ▷ Randomly sample data with batch-size datapoints
    $\mathbf{g}_t = \nabla f(\theta_t(\mathbf{a}_t), \mathbf{y}_t) + \theta_t \gamma$
    $\mathbf{v}_{t+1} = \mathbf{v}_t \mu + \mathbf{g}_t$
    $\theta_{t+1} = \theta_t - \alpha \mathbf{v}_{t+1}$
    $t = t + 1$
  end while
```

Momentum makes more sense when first having discussed the types of "loss landscape" of many problems expected to be solvable by ANN's. The loss landscape is in this case a colloquial term for the curvature of the loss function. Due to the high-dimensional problem statements most ANN's are expected to solve, the loss landscape may be filled with regions of highly changing curvature and a plethora of local minima and maxima (valleys and peaks).
The idea of momentum is then to break past these fluctuations in the landscape, instead of being forced to painstakingly traverse the landscape as it appears. The mechanism by which this works in theory, as explained in [30], is that repeated directions in the gradients will accumulate, and thus the update is more invariant to sudden changes in local curvature.
Therefore, the algorithm tends to not get stuck in local valleys and peaks, i.e regions of high curvature tend to not be reinforced in the momentum vector, while regions of low curvature tends to reinforce. Notice that the momentum vector is an analog to an exponential moving average estimation of the first moment (SMA is analytically similar to EMA from [9] section 3.2).

In the current literature, there has been shown special interest to methods that incorporate a concept called adaptive learning rates. The most popular variant of these algorithms are called Adam. Adam is based on the first statistical moment and the raw second statistical moment, and also assumes a normal distribution in the gradients for the assumption that these moments are correct. What follows is the variant called AdamW, which we present as algorithm [5]. Typically, the same type of weight-decay - as seen in the proposed SGD algorithms - is used in the original Adam paper [31], but as shown in [32], the weight-decay term as defined in [31] is not equal to $L2$ regularization and therefore this decoupled weight-decay term is presented instead.

Algorithm 5 Adam with decoupled weight-decay, AdamW

```
Require: $\theta$ = model parameters, $\alpha$ = learning rate, $t = 0$, $batch - size$ = Number of samples per training loop, $\gamma$ = Weight-decay ratio, $\beta 1, \beta 2$ = Exponential decay rates for the moment estimates, $\epsilon$ = numerical stabalizer
Ensure: $\theta \in \mathbb{R}$, $\alpha \in (0,1)$, $batch - size <= |data|$, $\gamma \in [0,1), \beta 1, \beta 2 \in [0,1), \mathbf{v}_0, \mathbf{m}_0 = \mathbf{0}$
  while $\theta \neq \theta^*$ do
    $t = t + 1$
    $\mathbf{a}_t, \mathbf{y}_t = rand(batch - size, data)$    ▷ Randomly sample data with batch-size datapoints
    $\mathbf{g}_t = \nabla f(\theta_t(\mathbf{a}_t), \mathbf{y}_t)$
    $\mathbf{m}_t = \mathbf{m}_{t-1}\beta 2 + \mathbf{g}_t(1 - \beta 1)$
    $\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{(1-\beta 1^t)}$
    $\mathbf{v}_t = \mathbf{v}_{t-1}\beta 2 + \mathbf{g}_t^2(1 - \beta 2)$
    $\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{(1-\beta 2^t)}$
    $\theta_{t+1} = \theta_t - \alpha(\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t + \epsilon}} + \theta_t\gamma)$
  end while
```

3.2.3 Temporal Data Analysis

A large amount of real world applications of data deal with tasks has a temporal component. This can be exemplified through a number of cases:

Consider the case of classifying the action of a person either picking up an object or not given a set of sequential state descriptions, where a state S, is simply a 1 or 0 depending on whether the person is holding an object or not. For simplicity, we will consider sequences containing two states only. The case [0, 1] then represents having picked up the object. Notice here that [1, 1] does not correspond to picking up the object, as the person already held the object in the initial state. Therefore, we see that the classification relies on the temporal transition. In this example, the input sequence might be of arbitrary length and contain cycles of picking up and putting down the object, which makes the significance of the transitional aspect of the temporal sequence higher.

Another example, which resembles the previous case, of temporal significance in data is captivated through the example selecting an action based on a sequence of previously recorded states. In contrast to classifying an action taking place in the sequence, we are now instead trying to select an optimal action which maximises some expectation.

Yet a third case can be thought of, in the case of taking a sequence of strings, eg. a sentence and defer meaning from it, based on the ordering of words and other grammatical tokens. In this case, the output sequence might also need to be of arbitrary length, such as generating a response to a question. In this case, both the input and output sequence does not have a predefined maximum depth. As such, we need a model architecture that can be made to handle such instances.

From these example cases, we see that the standard feed-forward definition of ANN's only gives us a workable activation if the input matrix is "flattened" or vectorized by concatenating all the rows into a single long row. As argued before however, this does not lend an obvious way to infer correct classification per the first example. Therefore, designs into ANN's types that deal with such data differently are a popular topic of research.

3.2.4 Recurrent Neural Networks

Recurrent Neural Networks are a type ANN's designed for solving tasks involving data with a temporal component. In the most simple setting, Recurrent Neural Networks process the combined input from the data sequence, with outputs from itself from previous "steps" in the data-sequence. Thus, given a data sequence consisting of $t = 1, 2, 3, ..., n$ different "frames", and a feature dimension f, we might have the following shape

$$\mathbf{D} = [\mathbf{t}][\mathbf{f}]$$

giving us a matrix whose rows represent the state of the input at a given time, and the cardinally of the columns gives us the features per row.

An RNN is then designed to extract information that is encoded into the data sequence. For a single RNN - computing cell in a larger network which consists of two separate activations - the following is a possible setup:

$$\mathbf{h}_t = \phi_h(\mathbf{W}_h \mathbf{a}_t + \mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{y}_t = \phi_y(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y)$$

where the sets of **W's** represent weight matrices for two activations in the cell - namely h and y. **U** - represents the weight that interact with the combined output of previous time-steps. Thus, for each i in the sequence, we build a representation at each current i, consisting of the combined outputs of previous states being used in the subsequent activations. This combined output from previous states is often referred to and stored as a "memory" vector, which can be used at each step i in the input sequence to serve as context for the current step. Thus, we have a reference to previous "states" of computation when using the RNN paradigm. The implication of this is that RNN's can be thought of as directed-cyclic graphs in contrast to ANN's which can be thought of as directed-acyclic graphs.

From this general framework of RNN, many independent definitions of what the RNN cell does at time = i can be formulated. Consider that an RNN definition might include a concept such as a context-window, which could be defined as doing something special with only a selection of time-steps at time i.

Examples of simple RNN models include the Elman Network, which was given as the simple RNN setup in [3.2.4] and Jordan networks that are describes by

$$\mathbf{h}_t = \phi_h(\mathbf{W}_h \mathbf{a}_t + \mathbf{U}_h \mathbf{y}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{y}_t = \phi_y(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y)$$

from which it can be seen that the major difference in the definition lies in passing either the internal hidden state - or if the final activation, namely $\mathbf{y}_t$ - will be stored and passed for the next time-step.

Thus it can be seen that from even this relatively low-component RNN, there is a lot of possible arrangements. One such arrangement can be multilayered RNN architectures. Consider the following stratagem: One RNN cell completely processes a sequence, outputting as a new sequence consisting of all $\mathbf{y}_t$ in the order of the computations. Then give this new sequence D' as input to the next layer. This gives the ability further propagate relevant temporal information to the final output. Furthermore, in the context of the Elman architecture both the $\mathbf{h}_t$ and $\mathbf{y}_t$ could serve as initial values of the hidden state and the new input sequence. This type of layering is often referred to as an encoder-decoder stack.

One problem that occurs with many models of RNN is the problem of vanishing gradients. Since we do not have a direct gradient flow before the RNN computations, the back-propagation algorithm must use the gradients purely propagated through the perhaps hundreds of forward-pass computations to compute the gradient. This can be seen as related to the vanishing gradient problem for very large models. Since we use the chain rule to perform back-propagation, we are calculating based on the earlier activations of the RNN cell itself. Thus, we might calculate the gradient many times before passing through a RNN cell completely. This both leads to the problem of vanishing and exploding gradient, as depending on the computations of the RNN cell, this gradient either gets larger or smaller through multiple applications of the chain rule, as the activation will rarely preserve gradient magnitude on the backward pass.

3.2.4.1 Long Short-Term Memory

The RNN architecture known as Long Short-Term Memory [44] - abbreviated as LSTM - is designed to address problems of vanishing gradients during the back-propagation algorithm. LSTM builds on the ideas of RNN's, by introducing "gates" of information. First, we give the full description:

$$\begin{aligned}
\mathbf{f}_t &= \sigma_g(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f)\\
\mathbf{i}_t &= \sigma_g(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i)\\
\mathbf{o}_t &= \sigma_g(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o)\\
\hat{\mathbf{c}}_t &= \sigma_c(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c)\\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \hat{\mathbf{c}}_t\\
\mathbf{h}_t &= \mathbf{o}_t \odot \sigma_h(\mathbf{c}_t)
\end{aligned}$$

Respectively, the gates can be described as the following Jordan-network-like activations ($\mathbf{h}_t$ in this case is the output of the final activation):

- The forget gate: $\mathbf{f}_t = \sigma_g(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f)$
- The input gate: $\mathbf{i}_t = \sigma_g(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i)$
- The output gate: $\mathbf{o}_t = \sigma_g(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o)$

Furthermore, another Jordan-like activation can be found for what is described as the cell input: $\hat{\mathbf{c}}_t = \sigma_c(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c)$, where $\sigma_c = tanh$ and $\sigma_g = sigmoid$ functions. Finally, these activations are combined as follows:

$$\begin{aligned}
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \hat{\mathbf{c}}_t\\
\mathbf{h}_t &= \mathbf{o}_t \odot \sigma_h(\mathbf{c}_t)
\end{aligned}$$

We see then that $\mathbf{c}_t$ - which is called the cell memory - is the sum of two computations. By looking at $\mathbf{f}_t \odot \mathbf{c}_{t-1}$, we see that this is the forget gate with the element-wise multiplication with the cell memory of the previous time-step. Therefore at $t = 0$, this part of the sum is $\mathbf{0}$. Intuitively, this corresponds to saying there is nothing to forget, as this is the initial part of the sequence.

The next part of the sum is the gated input of the current sequence step, and is the element-wise product of the two Jordan-like activations $\mathbf{i}_t$ and $\hat{\mathbf{c}}_t$. This is what defines the cell memory's computation for the current time-step.

At the final activation, the element-wise product of the output gate and current cell memory is taken. Therefore, the cell memory represents the gated information of previous time-steps, as it is the sum and combination of these and the output gate represents the information than can leave the current computation loop.

Taking a look at this computation path from the perspective of the back-propagation algorithm, there are now multiple "gradient paths" of computation, i.e. more independent partial derivatives to compute the gradient out of. Therefore, the collective gradient of partial products does not simply go to multiply along the same computation path when unrolled: this is a possible explanation of LSTM's ability mostly avoid vanishing gradients. Also, since LSTM in theory has the ability to gate information, arbitrarily long sequences become more plausible to make predictions from, as there is not a reliance on a single recurrent activation, but multiple - relatively independent - computations and information, not needed for further computations can be "forgotten". Also, both $\mathbf{c}_t$ and $\mathbf{h}_t$ can serve as initial cell memory and initial input sequence respectively for a multi-layer LSTM architecture.

3.2.5 Attention based models

Attention based models originated with the third case of temporal data analysis that we presented in mind. The rough idea is being able to assign importance to different words in a sentence, e,g, learn to pay "attention" to the most significant parts of a given input sequence.

A common application is tokenizing the sentence, by assigning specific numerical values to specific words. An example of this could be the word "Hello" being assigned the specific numerical value of 255. As such, a sentence of strings can be encoded as a sequence of integers or even floating point numbers. In most real cases, the embedding of a word is instead a vector of numbers in the reals. The idea of Attention is then, that each token vector can be calculated at the same time - looking at an arbitrary context window - compared to the normal recurrent structuring, where each token must be forward propagated through time to receive it's computation.

Attention also often uses sub-networks to represent the ability to give context to the word from different parts of the sentences. A common setup, is one sub-module encoding the "query", i.e. the important word itself as a vector, while another module might encode the "key" of the token and a final sub-module encoding the "value". This is combined in the end as:

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})\mathbf{V}$$

where d_k is the dimensionality of the word token. This is the typical computation of a transformer-encoder called *scaled-dot-product Attention.* We see here that since the values and importance of the different tokens does not rely explicitly on the forward-propagation through time - as is the case in RNN's - and therefore does not suffer from the effects of repeated chain-rule usage. Also, since computations of "importance" are performed in parallel across the entire input sequence, there is no longer a bottle-neck of sequential computations.

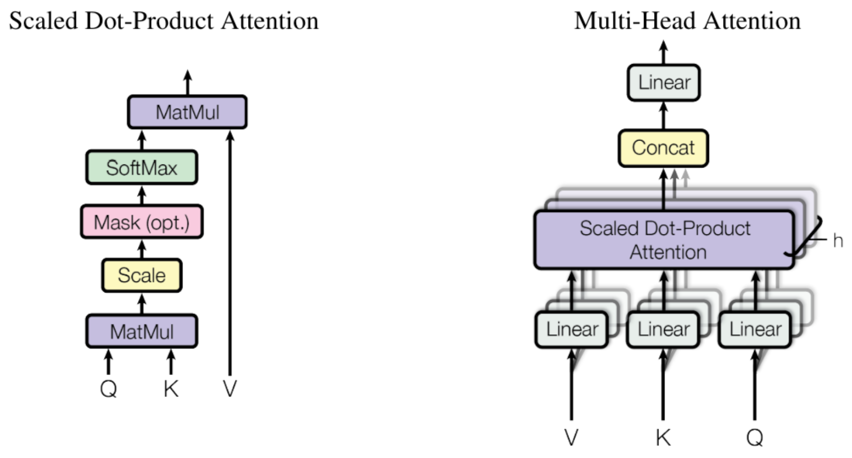

Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Figure 2: Attention mechanisms

3.2.5.1 Transformers Transformers are a novel invention found in the paper "Attention Is All You Need" [33]. The efforts of the paper showcase how a clever combination of Attention mechanism - especially multi-head Attention and scaled-dot-product attention - can be combined to get state of the art results for most machine translation tasks, entirely without the need for RNNs.

One of the important contributions of the transformer encoder is the inclusion of a multi-head Attention mechanism. In this setting, a number $h \in [1, n]$ where n is a positive integer greater than 1, of independent Attention "heads" are computed in parallel for the input sequence. As such, the reliance on one Attention mechanism is subverted for smaller - relative to number of weights - Attention heads, which are combined in the end. We borrow the figures from [33], which illustrates the ordering of computations and present them as [2].

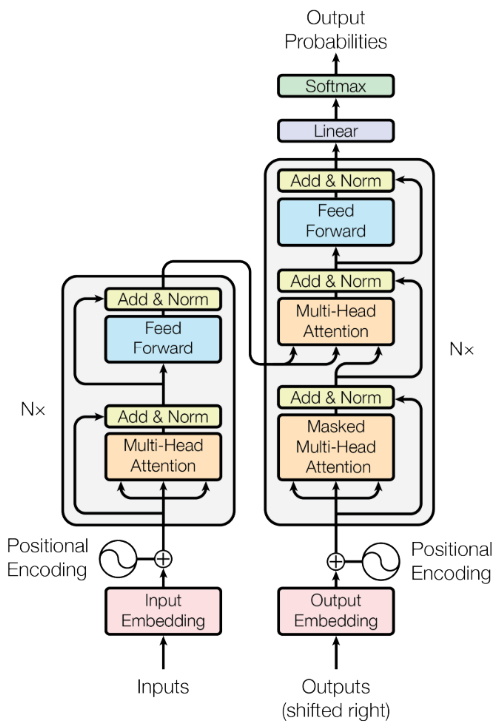

Figure 3: Transformer model

Furthermore, the Attention paper proposes applying a feed-forward ANN to each position of the input sequence. In accordance with the encoder-decoder paradigm, complete transformer models are based on the idea of having one section of the model ”encode” information from the input sequence, and a ”decoder” stack, which takes the output of the decoder and uses it as it’s initial sequence. In the transformer setting, this paradigm is further modified according to the figure which we borrow from [33] as [3]. As can be seen, several ”skip” connections throughout the model allows for better gradient back-propagation, as presented by [34].

3.2.6 Deep Reinforcement Learning

When discussing the second case of temporal data analysis, we framed the problem as maximizing the expected return. The question then becomes how to train an agent to this purpose, or rather, how to define a loss function such that the agent will learn to improve. The answer comes to us in the form of Reinforcement Learning.

We will present a similar problem as that given in [35] chapter 2.1. You are faced at multiple time-steps $t \in \mathbb{N}^+$, with $k \in \mathbb{N}^+$ with choices or actions, after which at each t, you are given a reward based on the action you took, and the state which the environment was in at the time. The reward is drawn from a stationary probability distribution. The goal is to maximize the reward over a certain period of t, where at each t, the current states possible is denoted S_t and the current possible actions are denoted A_t. This problem is commonly referred to as the k-armed bandit problem, as is the more formal version of the second case presented in our temporal data analysis chapter. We say that the value of choosing an action in a given state s_t, denoted $q^*(s_t, a_t)$, is the expected return of selecting an action a_t at state s_t:

$$q^*(s_t, a_t) = \mathbb{E}[R_t | A_t = a_t, S_t = s_t]$$

We say here that $q^*(s, a)$ is the "true" or optimal state-action value. At the start of learning, we start out with an estimate of $q^*(s, a)$ that may be entirely random, which we call $Q(s, a)$. The goal is to minimize the difference between our estimation $Q(s, a)$ and the optimal state-action value $q^*(s, a)$. We note that this formulation with the addition of the notion of a state, transforms the k-armed bandit problem into the finite Markov decision processes problem as presented in [35] chapter 3.1. In our presentation, we have presented the value as an estimation of the reward from the current t. It should be noted that other definitions are commonly used, based on the accumulated rewards in the future horizon called the return G_t. Also note that we have suffixed the realization of specific states and actions with t. This is to emphasise that a current state s_t might be differentiated from another similar state, by having occurred at a specific t.

What further defines the problem of finite Markov decision processes - which we will simply refer to as MDP - is the notion of the next-state. The logic is that the value of the current state is related to the value of potential future states, and not only the reward found for taking a given action in the current given state. A given characteristic or the "Markov property", is that for the analysis, it is assumed that the current state s_t contains the necessary information from all previous states that would be relevant for the decision at the current point.
Thus, the analysis of MDP's is focused with calculating on present states, future states and rewards, and not past ones. We argue that the notion of a specific state only occurring at a specific t is still compatible with the Markov property, since it can still contain all the necessary information for the current decision, but that two similar states be differentiated at the point they occurred.

Given this problem definition, we see that the problem becomes adjusting $Q(s_t, a_t)$ such that it approximates $q^*(s_t, a_t)$.

This is the problem that the family of algorithms known as reinforcement learning algorithms are designed to deal with. These agents are defined in terms of a policy π, such that taking an action is defined by the probability of taking an action a_t in a current state s_t: $\pi(a_t | S = s_t)$. Reinforcement learning algorithms learn to approximate $q^*(s_t, a_t)$ through repeated interaction with the environment. The most common way of computing the expectation that defines $q^*(s_t, a_t)$ is through the Bellman equation, which we provide with slight abuse of notation:

$$Q_\pi(s_t, a_t) = \Sigma_a \pi(a_t|S = s_t)\Sigma_{s_{t+1},r} p(s_{t+1}, r_t|S = s_t, A = a_t)[r_t + \gamma V_\pi(s_{t+1})]$$

It is important to note that the intention of this equation is for it to be a recursion equation, such that by iterative application, it will progressively learn better approximations of $q^*(s_t, a_t)$. Here we introduced a new term $V_\pi(s_{t+1})$. This term represents the value of the subsequent state under the current policy. Thus, the value of taking an action given the current state is dependent on $V\pi(s_{t+1})$, which is not explicitly defined at this point and the reward from the current t. Due to the yet defined nature of $\pi(s_{t+1})$, there is a wide array of possibilities on how to define it, which will lead to different approximations of $q^*(s_t, a_t)$.

Now that we have a way to define an "update" to our approximation, we can translate this into a loss function in use for ANN training. Consider the value of $Q_\pi(s_t, a_t)$ after an update. This value will be representing a closer approximation to the true state-value. We can then differentiate between this new approximation of $Q_\pi(s_t, a_t)$ and the old one. Thus given $x_t = Q_\pi(s_t, a_t)$ and $y_t = \Sigma_a \pi(a_t|S = s_t)\Sigma_{s_{t+1},r} p(s_{t+1}, r_t|S = s_t, A = a_t)[r_t + \gamma V_\pi(s_{t+1})]$, for a batch of values of x_t and y_t, then: $L(\mathbf{x}, \mathbf{y})$ can be computed using methods such as MSE. This now gives the ability to compute a gradient efficiently through a setup, which we showcase through [6].

Algorithm 6 Deep Reinforcement Learning framework

```
Require: $\theta$ = Initial parameters of model, $t = 0$, $batch - size$ = Number of samples per training
    loop, $\gamma$ = emphasis on future state, envi = transition model of the environment, $L$ = differentiation metric to be used eg. MSE.
Ensure: $\theta \in \mathbb{R}$, $|batch - size| <= |data|$, $\gamma \in [0, 1)$
  $data = \mathbf{0}$
  while $Q_\pi(s_t, a_t) \neq q^*(s_t, a_t)$ do
      $s_0$ = envi.reset()
      $t = 0$
      while !envi.done do
          $a_t = argmax(f(\theta_t(s_t)))$
          $s_{t+1}, r_t, done = envi.transition(s_t, a_t)$
          $data_t = (s_t, a_t, s_{t+1}, r_t, done)$
          if $|data| > batch - size$ then
              $s_b, a_b, s_{b+1}, r_b, done = rand(batch - size, data)$    ▷ Randomly sample data with batch-size data-points
              $\mathbf{x}_t = Q(s_b, a_b) = f(\theta_t(s_t))[.][a_b]$ ▷ Here the scripting after the function is meant to say that we select elements in the columns according to the actions
              $\mathbf{y}_t = \Sigma_a \pi(a_b|S = s_b) \Sigma_{s_{b+1},r} p(s_{b+1}, r_b|S = s_b, A = a_b)[r_b + \gamma V_\pi(s_{b+1})]$
              $L_t = L(\mathbf{x}_t, \mathbf{y}_t)$
              Calculate gradient of $L_t$ and back-propagate
              $\theta_{t+1} = optim(\theta_t, \nabla L_t)$
          end if
          $t = t + 1$
      end while
  end while
```

4 AttentionSplit

AttentionSplit is one of the main contributions of this paper. AttentionSplit is based in the idea that while Transformers effectively solve machine translation and/or text generation tasks - even with a new kind of transformer called visual transformers - there is still a time and a place for recurrent neural network architectures. Through this section of this paper, we shall first argue for once again combining ideas of Attention with ideas of RNN's. We discuss the specific cases where we have inclinations of potential success for this idea. We will then discuss combining "flows" of Attention from different parts of this module setup.

First we will present our model AttentionSplit in [7], and then break down and discuss its individual parts. We will explain the use of notation, which deviate from the normal operators. In the first line, when we reshape D, this can be seen as partitioning the data such that we have new dimensions. From this partitioning, it can be seen that $h \neq |\mathbf{t}|$, then there will be a blend of the elements of different "frames" in the sequence. Due to the nature of partitioning, the sequence length need not be pre-determined and the same weights can be used to calculate activations for arbitrary sequence lengths: the amount of "Attention heads" h is what determines the number of inner-products, which remains constant.

We apply layer normalization [37] and a non-linear activation function GELU [36] to the matrix multiplication and obtain $\mathbf{X}$, which represents our "Attention" encoding, which we explain in the subsequent sub-chapter. If there is a previous hidden state matrix, we apply a separate "Attention" encoding with different weights to this matrix. We then reshape $\mathbf{X}$ such that it represents a new partitioning. This partitioning requires works, since we create $\mathbf{W}_h = [h][\frac{dh}{|\mathbf{f}|}]$, such that $\mathbf{X}$ will have shape $[bs \cdot |\mathbf{t}| \cdot |\mathbf{f}|][\frac{dh}{|\mathbf{f}|}]$, we have sufficient components to partition the tensor into $[bs][|\mathbf{t}|][h][j]$, where $j = \frac{d}{h}$. We can think of this partitioning as rearranging the elements back into the temporal sequence, but partitioning now only the feature dimension into h equally sized sub-vectors in the overall tensor $\mathbf{X}$. If the hidden matrix is also present, we apply a similar partitioning. Next, we instantiate three tensors with similar shapes as to the newly partitioned $\mathbf{X}$, though as can be seen, one does not include the temporal dimension $\mathbf{t}$

The while loop is the recurrent part of out module. We use $\hat{\mathbf{C}}$ to represent the cell memory of previous time steps. Taking a look at the first few lines inside the while loop, first a dropout function [38] is applied to $\hat{\mathbf{C}}$. The reasoning is two-fold, namely that we want there to be a natural "Forgetfulness" associated with the hidden state. Typically, we apply 10% dropout, so we reason that only features reinforced across the temporal dimension will carry forward. The second reason is the regularization benefits argued in [38]. Next, if the $\mathbf{H}$ tensor is not $\mathbf{0}$, we will index the temporally corresponding element of $\mathbf{H}$ and take their sum. This carries the implication of creating a blend of information from previous cell memory states. There are then two possible scenarios: either $\mathbf{H}$ is computed at a previous time step of the environment and therefore $\mathbf{H}$ will contain the blended information from the previous time steps, and the second case is the encoder-decoder scenario where $\mathbf{H}$ is created during the same time step in the environment by a previous AttentionSplit layer. We can then use this information to reinforce the cell memory instead of it having a "cold start".

The next step is the definition of $\mathbf{C}_i$, which represents our cell memory for the current time step.

Algorithm 7 AttentionSplit forward pass

```
Require: $\mathbf{D} = [\mathbf{bs}][\mathbf{t}][\mathbf{f}]$, $\mathbf{H} = [\mathbf{bs}][\mathbf{t}][\mathbf{f}] \vee \mathbf{0}$, $bs$ = batch-size, $\mathbf{t}$ = time dimension of input tensor,
  $\mathbf{f}$ = feature dimension, $h$ = number of Attention heads, $d$ = hidden dimension, $\mathbf{W}_h = [h][\frac{dh}{|\mathbf{f}|}]$,
  $\mathbf{U}_h = [h][h]$, $\mathbf{A}$ = the cell weights having shape $[h][\frac{|\mathbf{f}|}{h}]$, $\mathbf{W}_f = [d][d]$, $\mathbf{b}_h = [d]$
Ensure: $bs \in \mathbb{N}^+$, $bs <= |data|$, $\mathbf{f} \in \mathbb{R}$
  Reshape $\mathbf{D}$, such that $\mathbf{D} = [bs \cdot |\mathbf{t}| \cdot |\mathbf{f}|][h]$
  $\mathbf{X} = \mathbf{GELU}(\mathbf{LayerNorm(DW}_h))$      ▷ $\mathbf{X}$ will have shape $[bs \cdot |\mathbf{t}| \cdot |\mathbf{f}|][h]$
  Reshape $\mathbf{X}$ such that $\mathbf{X} = [bs][|\mathbf{t}|][h][j]$
  if $\mathbf{H} \neq \mathbf{0}$ then
    Reshape $\mathbf{H}$, such that $\mathbf{H} = [bs \cdot |\mathbf{t}| \cdot |\mathbf{f}|][h]$
    $\mathbf{H} = \mathbf{GELU}(\mathbf{LayerNorm(HU}_h))$
    Reshape $\mathbf{H}$ such that $\mathbf{H} = [bs][|\mathbf{t}|][h][j]$
  end if
  $j = \frac{d}{h}$
  $\hat{\mathbf{C}} = [bs][h][j]$
  $\mathbf{CW} = [|\mathbf{t}|][bs][h][j]$
  $\hat{\mathbf{Z}} = [|\mathbf{t}|][bs][h][j]$
  $i = 0$
  while $i \neq |\mathbf{t}|$ do
    $\hat{\mathbf{C}} = \mathbf{DROPOUT}(\hat{\mathbf{C}})$
    if $\mathbf{H} \neq \mathbf{0}$ then
      $\hat{\mathbf{C}} = \hat{\mathbf{C}} + \mathbf{H}_{[.][i][.][.]}$      ▷ Here we mean selecting the $i$'th element of $\mathbf{H}$, which is a matrix,
  and performing element-wise addition
    end if
    $\mathbf{C}_i = \mathbf{GELU}(\mathbf{A}_{nhi}\mathbf{B}_{bhj}) \rightarrow \mathbf{C}_i = [b][h][j]$      ▷ In practice, the n dimension of $\mathbf{A}$ is artificially
  added, such that the einsum is applied to all batches
    $\hat{\mathbf{Y}}_i = \mathbf{GELU}(\mathbf{LAYERNORM}(\mathbf{X}_{[.][i][.][.]} + \mathbf{C}_t))$
    $\mathbf{WC}_{[i][.][.][.]} = \hat{\mathbf{Y}}_i$
    $\hat{\mathbf{C}} = \mathbf{SELU}(\hat{\mathbf{Y}}_i)$
    $\hat{\mathbf{Z}}_{[i][.][.][.]} = \hat{\mathbf{C}}$
    $i = i + 1$
  end while
  reshape $\mathbf{CW} = [bs \cdot |\mathbf{t}|][d]$
  $\mathbf{O} = GELU(\mathbf{W}_f^\top \mathbf{CW} + \mathbf{b}_f)$
  reshape $\mathbf{O} = [bs][|\mathbf{t}|][d]$
  $\mathbf{H} = \hat{\mathbf{Z}}$
  return $\mathbf{O}, \mathbf{H}$
```

We apply an einsum, such for each partition h of the feature dimension, will give a product with $\mathbf{A}$, such that each partition h of $\mathbf{C}_i$ will make a product with exactly one of h rows of $\mathbf{A}$. In practice - because of the batch dimension of the input data - we artificially "append" the extra dimension n to $\mathbf{A}$, though this is a singular dimension. Therefore, the same matrix $\mathbf{A}$ is applied the same to all batch elements. We apply a layer normalization and non-linear activation of $GELU$ to this computation. At the next line, we take the sum of the corresponding temporal element of $\mathbf{X}$ with $\mathbf{C}_i$. The idea here is that we propagate relevant information through the time steps.

Before continuing to the next time step, we save $\hat{\mathbf{Y}}_i$ in the corresponding ith row of $\mathbf{CW}$. We then apply a $SELU$ non-linear activation to $\hat{\mathbf{Y}}_i$ and save as the cell memory of previous time steps for the next i. It could be perceived as strange to apply a second non-linear activation right after the $GELU$. The reason for having a $SELU$ [39] activation right before saving this as the cell state for previous time steps, is to ensure extra normalization. If normalization is not ensured, then for a large enough amount of time steps, we might get values propagated that become very large. The values getting too large can even in some cases cause values to become impossible to compute in practical scenarios. As argued in the $SELU$ paper, this particular non-linear activation will prevent any values of becoming too large in either positive or negative directions. This value is also saved in the tensor $\hat{\mathbf{Z}}$.

After the recurrence, we apply a final feed-forward activation to each temporal row feature dimension in the stored "frames" of $\mathbf{CW}$. We assign $\mathbf{H} = \hat{\mathbf{Z}}$ as the cell memory matrix for future layers and reshape $\mathbf{CW}$ into a temporal sequence similar to the input sequence, but now encoded by the AttentionSplit layer.

To unite some notification from pseudo-code practices with mathematical notations, we have sacrificed notations of dimensionality for a more "functional" explanation, closer to a real implementation in code. We also use set notation to indicate the number of components in a vector / matrix, which may conflict with regular notations of vector / matrix norms.

4.1 Combining Attention with Recurrence: A New Perspective

In this work, we have used a definition of Attention that is a not typically the one found in most literature on the subject. Instead, we use a definition inspired by the concept of the Attention found in [33]. In the paper, the idea is based around learning to assign attention to important words for a sentence, i.e. learn which parts of the sequence are the important parts, instead of trying to do this by propagating the information through recurrence. This approach is what inspired the AttentionSplit partition. In AttentionSplit, we also try to identify important parts of the sequence. This is done by a partitioning of the input sequence, such that individual temporal rows, will have their information "blended" together. We then compute inner products with a weight matrix that has rows corresponding to each part of the partition of the input sequence. This is obviously a conceptually and computationally simpler type of attention as compared to the scaled dot-product and multi-head Attention, and shares only perhaps the origin in the same idea.

The conception behind the attention of transformers primarily focus on textual data - such as sentences and words - whereas AttentionSplit tries to incorporate the same concept for arbitrary temporal sequences. AttentionSplit tries to give a new perspective to combining the ideas attention with recurrence, by first applying attention encoding that seeks to blend important information of different temporal rows, and then using recurrence to propagate this to the forwards most layer. This approach makes more sense if thinking about the first case from [3.2.3], where we proposed the scenario of classifying picking up objects. First we apply attention to identify important sequence parts, then we apply recurrence to propagate this information to the forward-most part of the sequence. This information can then be mapped in an external layer to a classification. This is the main reasoning behind once again combining attention with recurrence, as there might be a combined benefit to both types of temporal encoding, especially when combined.

Another perspective on why combining attention with recurrent structures can be given when considering applying attention to previous matrix bundles of cell memory vectors. This allows the power of identifying important sequential part of past cell memories as can be done for the input sequence. We believe this is an area of research that deserves more in-depth exploration. Applying attention to the previous cell memory might provide insightful information to subsequent layers. We have designed AttentionSplit such that it can work either with or without such a matrix.

4.2 Recurrent Component of AttentionSplit

The recurrent component relies on combined output of previous cell memory states, with the output of the corresponding temporal component of the attention encoding of the input sequence. As such, this can be seen as most like a Jordan-like recurrence, and as thus is one of the simple recurrent modules that we discussed in theory section. In comparison with LSTM, a much simpler recurrent setup is present, although it shares the notion of forgetting particular information, though instead of a learned forget gate, we apply heuristic dropout, and rely on a ”momentum” in the cell memory to reinforce the important features. Therefore, AttentionSplit could most likely be improved by using more clever recursion, with gated units in similar composition to that of LSTM. Therefore, along with more sohpisticated applications of attention such as multi-head attention, scaled dot-product attention and self attention could greatly improve upon the performance and theoretical soundness of the AttentionSplit module. In fact, we will go so far as to call AttentionSplit a ”bare-bones” representation of the potential possible combinations of attention and recurrence. We merely hope that the architecture can serve as a point of interest to draw further research into the combination of these two paradigms of temporal data analysis.

4.3 Combining *flows* of Attention

From the way we compute an encoding of the input sequence, we see that the cell memory gets a similar attention encoding. In the recurrent part, we then combine this attention encoding with the current cell memory at each step, before combining it with the encoding of the input sequence. We thus see that we have two different attention encodings combined. One containing the flow of attention as per a previous time step, or another encoding altogether of the input sequence. We here give precedence to the input encoding, as the previous cell memory might ”wash out” information in the input encoding that is important to forward propagate. Therefore, we combine it with the current cell memories before this is taken through the mapping with the cell weights.

5 OrthAdam

5.1 Background in AdamP

We present OrthAdam, which is an Adam optimizer variant heavily inspired and based on the ideas presented in [40]. To summarize, AdamP is one of two optimizers found in the paper, which deal with the effect of momentum on scale-invariant weights. The paper shows that for the scale-invariant weights introduced by the varies methods of normalization regulation, there is a rapid-decay of effective step-sizes. This is due to fact that normalization introduces a scale-invariant forward and backward pass, which will introduce this scale-invarience into the gradient calculation. Said paper goes on to show in section 2.4 how this will lead to rapid decay of effective step-sizes. OrthAdam concurs with the findings of the paper, but propose that for the Adam variants of the proposed orthogonal projection, the following criteria be used:

$$\mathbf{q}_t = \begin{cases} \prod_{\mathbf{w}_t}\left(\frac{\hat{\mathbf{m}}_t}{\sqrt{(\hat{\mathbf{v}}_t+\epsilon)}}\right) & cos\left(\mathbf{w}_t, \frac{\hat{\mathbf{m}}_t}{\sqrt{(\hat{\mathbf{v}}_t+\epsilon)}}\right) < \frac{\delta}{\sqrt{dim(\mathbf{W})}} \\ \frac{\hat{\mathbf{m}}_t}{\sqrt{(\hat{\mathbf{v}}_t+\epsilon)}} & otherwise \end{cases}$$

The presented change of criteria is the main contribution of this part of the work. While we have named our proposed change as OrthAdam, this is merely to distinguish it from the original projection proposal of AdamP, and not to imply an increased sense of novelty. We fully acknowledge this as an incremental step further in the ideas presented by [40].

5.2 Change of Projection Criteria: Intuition

We intuitively think about using changing the criteria of projection in the following way: the basis gradient is a reference to the directions maximal change, calculated based on the current batch sample. At any moment, parameters of the model might be close a local peak or valley, and as such, the gradient will reflect this loss surface. Projecting - in the case - gradients reflect a "messy" loss surface, might lead to projecting in cases that are as such undesirable, as the key idea of keeping the direction of orthogonality in these cases could lead to increased sensitivity to local loss landscape structure, which is what momentum classically is designed to deal with. Clearly the authors of [40] show that their projection criteria will lead to increases in generalization ability across the board in many tasks, however, this work finds that using the projection term proposed in this work can lead to even further improvements across a selection of vision tasks.

5.3 Change of Projection Criteria: Justification

The main inspiration behind the change of criterion is the proposed scale-invarience of Adam asserted by [31] section 2.1:

$$\frac{c \cdot \hat{\mathbf{m}}_t}{c \cdot \hat{\mathbf{v}}_t} = \frac{\hat{\mathbf{m}}_t}{\hat{\mathbf{v}}_t}$$

if the Adam basic update is itself scale-invariant, we propose that this satisfies the orthogonal projection criteria, since this scale invarience should not lead to the weight norm growth as proposed by [40] as lemma 2.2. This all also relies on the fact that Adam's update is not - in our opinion - fully encapsulated by the assumption of equation 7 seen in [40], which is the basis for the argument. We present our findings in a selection of vision tasks in later chapters, and show that a change in projection criteria in the case of Adam's scale-invariant update, can potentially lead to small increases in generalization ability.

Algorithm 8 Proposed OrthAdam Algorithm

```
Require: $\theta$ = model parameters, $\alpha$ = learning rate, $t = 0$, $batch - size$ = Number of samples
  per training loop, $\gamma$ = Weight-decay ratio, $\beta 1, \beta 2$ = Exponential decay rates for the moment
  estimates, $\epsilon$ = numerical stabilizer
Ensure: $\theta \in \mathbb{R}$, $\alpha \in (0, 1)$, $batch - size <= |data|$, $\gamma \in [0, 1), \beta 1, \beta 2 \in [0, 1), \mathbf{v}_0, \mathbf{m}_0 = \mathbf{0}$
  while $\theta \neq \theta^*$ do
    $t = t + 1$
    $\mathbf{a}_t, \mathbf{y}_t = rand(batch - size, data)$          ▷ Randomly sample data with batch-size datapoints
    $\mathbf{g}_t = \nabla f(\theta_t(\mathbf{a}_t), \mathbf{y}_t)$
    $\mathbf{m}_t = \mathbf{m}_{t-1}\beta 2 + \mathbf{g}_t(1 - \beta 1)$
    $\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{(1-\beta 1^t)}$
    $\mathbf{v}_t = \mathbf{v}_{t-1}\beta 2 + \mathbf{g}_t^2(1 - \beta 2)$
    $\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{(1-\beta 2^t)}$
    $$\mathbf{q}_t = \begin{cases} \prod_{\mathbf{w}_t}(\frac{\hat{\mathbf{m}}_t}{\sqrt{(\hat{\mathbf{v}}_t+\epsilon)}}) & cos(\mathbf{w}_t, \frac{\hat{\mathbf{m}}_t}{\sqrt{(\hat{\mathbf{v}}_t+\epsilon)}}) < \frac{\delta}{\sqrt{dim(\mathbf{W})}} \\ \frac{\hat{\mathbf{m}}_t}{\sqrt{(\hat{\mathbf{v}}_t+\epsilon)}} & otherwise \end{cases}$$
    $\theta_{t+1} = \theta_t - \alpha(\mathbf{q}_t + \theta_t\gamma)$
  end while
```

6 Models

6.0.1 Soft Actor Critic

The Soft Actor-Critic (SAC) model is a reinforcement learning algorithm which aims to balance maximizing the expected cumulative reward, and achieving an optimal level of policy conformity - exploration vs. exploitation - which aids in avoiding premature convergence to sub optimal policies, and increases the algorithms resilience to unpredictable environments. One of the primary features of SAC is its use of a clipped double Q-learning approach, which reduces the overestimation bias often found in Q-learning, leading to improved stability and sample efficiency. SAC efficiently uses past experiences stored in a replay buffer, making better use of these experiences as an off-policy algorithm, than on-policy algorithms which discard experiences after a single use. Because of these features, SAC is very effective for tasks which consist of continuous action spaces [42]

In the introductory paper on SAC, this algorithm was used on a plethora of OpenAI Mujoco environments such as HalfCheetah and Walker2D. As reasoned by this, we also use SAC agents on said Mujoco environments in this project, as to compare a typical implementation to one in which the AttentionSplit module has been added to the agent's actor-architecture. This allows for fair comparisons between our results, and gives us insight into the potential efficacy of adding AttentionSplit to other contemporary techniques.

6.0.2 Deep Q-Learning

We describe our use of a Deep Q-Learning agent in [7.4], however we will here briefly describe what Deep Q-Learning is, and how it relates to contemporary research:

Looking back at our definition of the deep reinforcement learning framework in [6], we had yet to give a definition of $V(s_{t+1})$. Deep Q-learning - also simply called DQL - is one such algorithm which will define $V(s_{t+1})$, such that the expectation of future rewards can be calculated. DQL proposes using $V(s_{t+1}) = max(Q(s_{t+1}, a_{t+1}))$, which means that DQL evaluates the future value found in s_{t+1} as the value of taking the "best action" in the next step when following policy π. This gives a simple yet powerful formulation, which can be used for calculating the loss for a scenario corresponding to case two in [3.2.3] or simply the problem of solving MPD's as summarized in the section on Deep Reinforcement Learning. Algorithm [9] showcases an implementation of DQL in the same framework which we used to showcase the general deep reinforcement-learning setting.

Algorithm 9 Deep Q-learning algorithm

```
Require: $\theta$ = Initial parameters of model, $t = 0$, $batch - size$ = Number of samples per training
  loop, $\gamma$ = emphasis on future state, envi = transition model of the environment, $L$ = differenti-
  ation metric to be used e.g. MSE, optim = optimizer algorithm to use e.g. AdamW
Ensure: $\theta \in \mathbb{R}$, $|batch - size| <= |data|$, $\gamma \in [0, 1)$
  $data = \mathbf{0}$
  while $Q_\pi(s_t, a_t) \neq q^*(s_t, a_t)$ do
      $s_0$ = envi.reset()
      $t = 0$
      while !envi.done do
          $a_t = argmax(f(\theta_t(s_t)))$
          $s_{t+1}, r_t, done = envi.transition(s_t, a_t)$
          $data_t = (s_t, a_t, s_{t+1}, r_t, done)$
          if $|data| > batch - size$ then
              $s_b, a_b, s_{b+1}, r_b, done_b = rand(batch - size, data)$     ▷ Randomly sample data with
  batch-size data-points
              $\mathbf{x}_t = Q(s_b, a_b) = f(\theta_t(s_b))[.][a_b]$ ▷ Here the scripting after the function is meant to say
  that we select elements in the columns according to the actions
              $\mathbf{y}_t = r_b + (\gamma \cdot max(f(\theta_t(s_{b+1})))$
              $L_t = L(\mathbf{x}_t, \mathbf{y}_t)$
              Calculate gradient of $L_t$ and back-propagate
              $\theta_{t+1} = optim(\theta_t, \nabla L_t)$
          end if
          $t = t + 1$
      end while
  end while
```

Our rationale behind choosing DQL over other potential candidate algorithms is mostly due to the greater abundance of usage in the literature that we discovered covering the subject. Specifically, we took great note of the results of [43], as well as the simple transition from theory to application of this setting for the off-policy variant of RL agents. Since DQL is off-policy, it does not impose restrictions on our technique of batching. Also continued re-evaluations of the same state batches could in theory lead to different evaluations of $\mathbf{y}_b$, even though the stored information remains the same. This is because the definition of V_b in DQL always chooses the maximal value based on the current policy evaluation.

7 Datasets

Before giving a description of each datasets, it is important to remember that throughout this paper, we propose not one, but two developments. *OrthAdam* by itself is as mentioned in [5] and optimizer, which as a concept is an intrinsic part of neural networks and thus is seen across all types of architectures and tasks, whereas *AttentionSplit* [4] is an implementation of an attention mechanism which is able to be plugged into other models. As a result of this, some datasets are not used across both concepts but rather for one of them individually.

OpenAI's Gymnasium API is designed specifically for reinforcement learning tasks, and made it so we did not have to implement and verify a custom environment. Gymnasium is stated to be designed as a *"community effort"* and works with large and established organizations such as NVIDIA and Amazon [13], and have been developed with the findings of current academic literature in mind, which reinforces the integrity of the framework and thus establishes a higher validity of the results produced in this paper.

7.1 OpenAI Gymnasium - Classic Control Environments

In this paper, we use the classic control environment *Acrobot-v1* - an environment which is based on the work of *Richard S. Sutton* [15]. The goal of this environment is to apply torque to a joint attaching two link with the purpose of swinging it above a certain height. With a three-value action space and an observation space of six. Thus, this environment can be considered relatively simple, which is one of the purposes of the classic control environments. We also use the environments: *Cartpole-v1* and *Mountain Car-v0*, where the same simplicity is also present.

7.2 OpenAI Mujoco Environments

Mujoco stands for *Multi-Joint dynamics with Contact* and functions as a physics engine. From this we use the Half Cheetah [17] environment based on the work of *P. Wawrzyński* [16] which consists of a two-dimensional robot with nine body parts and eight joints, and has to purpose to create forward movement by applying torque to said joints. We further use the Walker2D environment which builds on the Hopper environment found in [41], and as with HalfCheetah, is generally regarded as a harder environment than that of the classic control environments, and thus gives us a larger spread of applications for which our proposed methods have shown results. We have combined the implementation ideas of [45] with the implementation of SAC to meaningfully integrate AttentionSplit into a model for playing the given Mujoco environments.

7.3 Image Classification & Object Recognition

As *OrthAdam* is an optimizer, we wanted to make sure that we did not just evaluate it's impact on reinforcement-learning and time-series applications, as its use-cases is much broader. Both the datasets *FMNIST* [20] and *CIFAR-10* [19] are well established and widely used within the academic literature. These datasets allows us to test the impact of the *OrthAdam* optimizer on tasks and architectures which have already been thoroughly investigated and in which small improvements - in the order of fractions of a percentage of accuracy - can represent considerable findings. Further, benchmarks of current state-of-the-art models are widely accessible and as has been established in other chapters in this paper, allows for better one-to-one comparisons to other methods. We further use the *Street View House Numbers* [21] and *CIFAR-100* [19] datasets for the same reasons.

7.4 Deep Q-Learning Predictions

In chapter [7.1] we introduced the OpenAI *Control* environment on which we performed model evaluations, however this introduction might not be fully forthright. Instead of solving these environment directly, we instead attempt to predict the outputs of a pre-trained Deep Q-Learning model [6.0.2], in essence, trying to predict how another state-of-the-art agent would play the game instead of the game it self. We propose this setting firstly because the common DRL algorithms typically already perform very well on these environments, and we therefore conjecture that the potential performance gains will be hard to evaluate and differentiate from those of the basis algorithms themselves.

Another reason we wanted to do a prediction series of another agent playing, is that we can take the data from the already trained agent, and then train other separate predictors of the agent's actions with the same exact data available to each predictor of the agent. We believe this further isolates the "true" performance metrics of the different predictor networks, as there is no chance that one network simply receives a harder prediction task, such as trying to predict a more sophisticated agent's action as compared to the potential other predictors.

For our experiment, we compare the prediction power of three separate networks, one with AttentionSplit components, one with a LSTM components and one with transformer encoder components, which share the following structure:

- Input feedforward. This first layer is to perform a learnable linear mapping, transforming the input dimensionality to a higher-one. The control environments of gymnasium often have low input spaces, so therefore, to allow for certain amount of "heads" for AttentionSplit and the Transformer-encoder
- Two stacked layers of the chosen recurrent or Attention type
- Feed-forward output layer, acting on the last item in sequence from the sequential encoders

Since we have only used the transformer encoder layers in these tests, we will explain the reasoning, which is twofold. Firstly, this would require further components to be added to only one of the networks to function, which would mean either extending the other two architectures, which would make the networks bigger than our practical means allows. Secondly, it is not immediately obvious how to generate the target sequence, when the target itself is only a singular label. Therefore, it could be said that this is not the ideal scenario for a full transformer stack, which is one of the motivations for AttentionSplit's design.

8 Machine Learning Operations

8.1 Data Preprocessing

8.1.1 Data Augmentation

Data normalization and augmentation are instrumental tools to increase stability and accelerate the training process of artificial neural networks. Furthermore, data augmentation is shown in multiple studies that it can lead to an increase in accuracy on a variety of models and data sets [1] [2]. A concrete example of a context where data augmentation shows to be useful, and which has been used throughout this paper, is the use of data augmentation to prevent over-fitting on complex image recognition tasks. Take the process of recognizing objects in images from datasets such as the Oxford Pets Dataset [4] where features such as lighting, backgrounds and camera angles could present biases towards potentially irrelevant features - seen especially in small datasets - which reduces the generalization ability of neural networks. In such a case, data augmentation such as random erasing [5], clipping, color-normalization and projections, can reduce the prevalence and impact of such features. Essentially one can say that data augmentation encapsulates the notion of extending the available data by creating similar inputs by altering the already existent data or creating completely synthetic data

Small datasets occur frequently in fields such as the medical industry, where both privacy concerns and rare diseases can limit data quantity.
The following is a list of various data augmentation techniques, as well examples of how we have incorporated a subset of these into this project:

Basic Image Manipulations:
Encompassing a large variety of techniques, many of these methods are generally well established as a result of their wide-spread use. Consider the CIFAR-10 and CIFAR-100 datasets, we utilized both *random flipping* and *random cropping*, which by looking at table [1] and figure [4], is shown to have a noticeable effect on the overall accuracy of our models. As a side note, the reason behind using these exact augmentations is based on the same choices and considerations in many papers throughout the academic literature [9], [10], [11] which allows for a potentially better comparability between other neural network types, and also allows us to utilize the data and results of these peer-reviewed papers, as to mitigate the possibility of faults in our own testing and evaluation of custom implementations.

The following list gives an overview of each configuration of transformations were used across all models and environments that were examined throughout this project:

- OrthAdam [5] on FMNIST [20] & SVHN [21]

  For FMNIST we purely use horizontal flipping, however we also needed to resize our data to $32x32$ as densenet requires a larger input size than FMNIST has. For SVHN we converted the images to greyscale as the only augmentation.

Many other types of data augmentation exists, and while the following techniques have not been used in this project, they are still valuable to mention.

Adversarial Training:
This term describes a technique in which a model attempts to accurately create realistic outputs - statistically similar outputs to that of the original training data - but is being continuously challenged by way of sabotage, with some alternative approaches. A well known architecture within this area is Generative Adversarial Networks, which in simple terms can be explained as two models, a generator and a discriminator, where the former attempts to generate output similar to its given input data [7].

Utilizing generative techniques, we can indirectly train the generator by updating the model - making it able to create more realistic outputs - by using the discriminators ability to discern real and fake generations from each other. The goal is to make the generator good enough, that the discriminator would not be able to distinguish the generated data from the original input data.

In the context of this project, it could be beneficial to measure the impact of combining adversarial training with the *AttentionSplit* module and the *OrthAdam* optimizer. as we have found throughout our research that adversarial training on attention-based models is a somewhat new concept.

Transfer Learning:
Utilizing a neural network model which is already trained on datasets for purposes which might not be the same as the neural network which is currently desired, can reduce the quantity of necessary data to gain high-quality model performance. Consider the scenario of two tasks: facial detection, and emotional classification based facial expressions.
Speculatively we could say that facial detection models should already be well-versed in identifying facial features such as eyes, lips and their relative positions, as these are highly correlated with whether or not a given input represents a face or not. It is fair to assume that the former model will have implicitly reduced the data-space into the most relevant features, and thus the latter model could avoid learning how to detect these features by itself, but rather bootstrap its own predictions using the newly created data.

Test-Time Augmentation:
A short note, which gives an insight into the possibilities of future work and verification of the results shown in this paper, is the paradigm of test-time augmentation. In this chapter, we describe the importance, and our implementation of data-augmentation. Test-Time Augmentation describes how augmenting data during the testing phase of a model can produce averaged outputs of models, helping to reduce noise and ambiguity. For models which are already sensitive to data-augmentation, test-time augmentation could help solidify the results of both *AttentionSplit* and *OrthAdam*, especially in cases with small datasets. [12]

Table 1: Impact of Augmentations on Model Accuracy (Using *OrthAdam*)

Model & Environment	**Mode**	**Accuracy (%)**
CIFAR-10 - Densenet121	Augments	90.35
	No Augments	84.24
CIFAR-10 - Resnet18	Augments	87.42
	No Augments	79.60
CIFAR-10 - Resnext50_32x4d	Augments	89.09
	No Augments	80.94

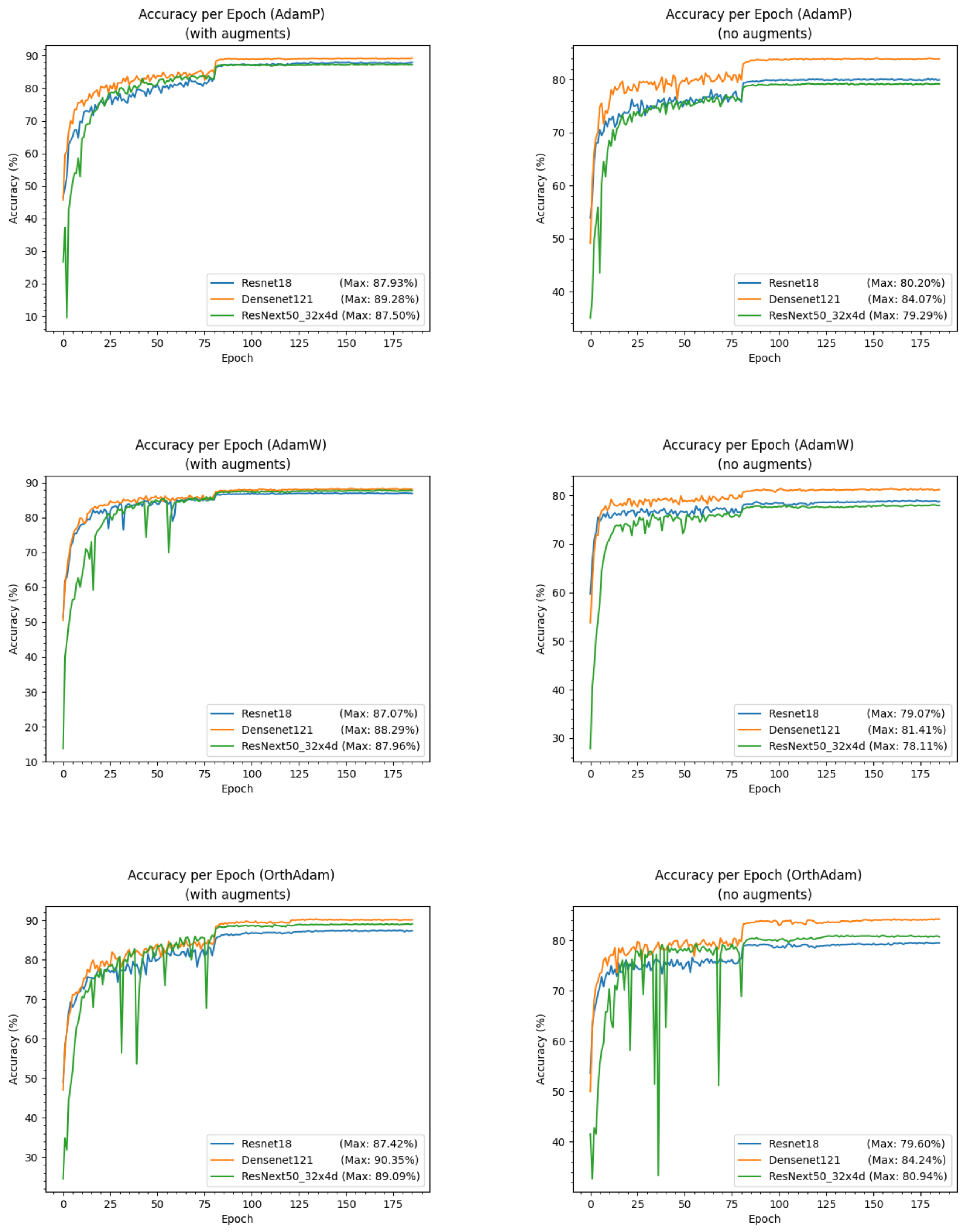

Figure 4: Impact of Augmentation on Model Accuracy (CIFAR-10)

8.1.2 Labelness & Preservation

Whenever synthetic generation or modifications are applied in the context of neural networks, there are many factors which should be thoroughly investigated for a project to be considered responsibly designed. One of the key issues that could arise is called *labelness*. Consider the classic task of - potentially hand-written - digit recognition. In this case, one has to be careful as transformations such as horizontal flipping could make it unfeasible to distinguish between letters such as b and d - depending on the font - and would cause their respective labels to become erroneous. A rather silly example would be reducing the color space of a model trying to classify or categorize data where said color-space would be the defining characteristic of the dataset.

In the context of this project, we spent a considerable amount of time both testing various transformation configurations, and deducing - by nature of each individual task - dangerous augmentations. Here it should be noted that these are to a significant extent speculations, as the computational requirements of performing a grid search across many configurations would be infeasible for us to perform. Since it would not be appropriate to give a comprehensive description of each and every consideration we made, we will highlight a few of them here:

- CIFAR-10

  As each image is standardized to be $32x32$ pixels in size, the resolution is already relatively low and while reducing the complexity and computational requirements of a given model, could be a detrimental to classification accuracy. At this scale, it is harder to discern unique characteristics of objects such as trucks and automobiles, since these labels already share many similarities and proper classification is contingent on minor details than between said classes and say, frogs. It could be interesting to asses the implications of using deep-learning based up-scaling techniques or context-based generative image expansion, but for the scope of this project, we decided to stick to simpler methods.

- Mujoco Half Cheetah

  In this environment, an agent applies torque to a joint with the goal of creating a forward movement by interacting with 16 other joints, each with varying but ostensibly high feature-dependence, could exactly because of this characteristic be significantly hindered if techniques such as data sub-sampling or temporal augmentation were used.

It is worth noting that currently there is no objective measure of labelness or if an augmentation definitively assures label preservation. Because of this, the standard practice is to perform some degree of label-validation by assessment from human agents.

8.2 Tools & Frameworks

8.2.1 PyTorch

For our implementation of both the *AttentionSplit* architecture and *OrthAdam* we utilized the *PyTorch* [29] library. PyTorch is a modern machine learning library built on the open-source *Torch* library and was released in 2016. We have found during the research phase of this project that PyTorch is the most widely used library in current machine learning research. Resulting from this decision, it is possible to use our implementations of the techniques mentioned throughout this paper through the PyTorch library, making it possible to - as with any other PyTorch neural network module - *"plug-and-play"* e.g. OrthAdam into models is you would with any other optimizer found in the framework.

Listing 1 Using OrthAdam in PyTorch Models

```
from modules.Optimizer  OrthAdam
densenet = densenet121(num_classes=10)
    densenet_criterion = nn.CrossEntropyLoss()
    densenet_optimizer = OrthAdam(...)
    densenet_scheduler = torch.optim.lr_scheduler.MultiStepLR(
        densenet_optimizer, [81, 122], 0.1
    )
```

As can be seen on listing [1], OrthAdam can be imported and instantiated semantically equivalently to e.g. AdamP or AdamW, as it is implemented via. inheritance from the `torch.optim.Optimizer` class. As we mentioned in chapter [2], the purpose of this paper is to propose techniques which we consider to be worthy of further exploration, and as such we found it important to allow further research to utilize an already existing and robust implementation of these techniques, instead of having to implement them from scratch using theoretical descriptions from chapter [4] and [5]. This lowers the barrier-of-entry, and helps ensure consistency throughout future research.

Furthermore, having a robust implementation of these techniques in a framework which sees use in the industry, would allow for evaluations of their efficacy in production-scale use as it eases the implementation process and thus would allow for more expansive testing of the efficacy of these techniques in large-scale real-world applications, instead of the relatively small-scale tests performed for this project. Listing [2] showcases how the AttentionSplit block can be seamlessly added to networks inheriting from PyTorch's `torch.nn.Module`'s

Listing 2 Adding AttentionSplit to PyTorch Neural Network Implementations

```
...
from modules.Attentionsplit import AttentionSplit

class Network(nn.Module):
    def __init__(self, actions):
        super(Network, self).__init__()
        self.actions = actions

        self.vision = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=11, padding=5),
            nn.BatchNorm2d(16),
            nn.GELU(),
            nn.MaxPool2d(3),
            nn.Conv2d(
                in_channels=16, out_channels=32, kernel_size=7, padding=3, groups=8
            ),
        ...
        self.encoder = AttentionSplit(256, 256, 8)
        ...

    def forward(self, states, hidden=None):
        ...
        states = states.view(-1, states.shape[2], states.shape[3], states.shape[4])
        x = self.vision(states)
        x = x.view(batch_size, num_frames, -1)
        x, c = self.encoder(x, hidden)
        return self.action(x[:, -1, :]), c
```

8.3 Training Pipeline

Because of this implementation, it was possible for us to use close to identical processes for testing and training our techniques as seen in papers such as [9], which helps to assert the validity of our results. As can been seen in listing [3] we use PyTorch's built-in train- and test loaders, which along with other features, fetches datasets automatically which mitigates the possibility of erroneous and inconsistent data. We describe these steps as we consider this project not only as a proposal for a topic which shows merit for future research, but also to detail how to use the described modules as a product, in practice.

Listing 3 Example of a Train- & Testing Loop

```
# Load the FashionMNIST dataset
trainset = torchvision.datasets.FashionMNIST(...)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=128, shuffle=True, num_workers=2
)

# Setup DenseNet
densenet = densenet121(num_classes=10)
    densenet_criterion = nn.CrossEntropyLoss()
    densenet_optimizer = OrthAdam(densenet.parameters(), lr=0.003, weight_decay=3e-4)
    densenet_scheduler = torch.optim.lr_scheduler.MultiStepLR(
        densenet_optimizer, [81, 122], 0.1
)

# Train Densenet
densenet_losses = []
densenet_accuracies = []
for epoch in range(args.epochs):
    densenet.train()
    for i, (inputs, targets) in enumerate(trainloader):
        inputs = torch.nn.functional.interpolate(inputs, size=(32,32))
        inputs, targets = inputs.to(device), targets.to(device)
        densenet_optimizer.zero_grad()
        outputs = densenet(inputs)
        loss = densenet_criterion(outputs, targets)
        densenet_losses.append(loss.item())

        loss.backward()
        densenet_optimizer.step()
    densenet_scheduler.step()


# Test Densenet
densenet.eval()
densenet_correct = 0
densenet_total = 0
for i, (inputs, targets) in enumerate(testloader):
    inputs = torch.nn.functional.interpolate(inputs, size=(32,32))
    inputs, targets = inputs.to(device), targets.to(device)

    outputs = densenet(inputs)
    _, predicted = torch.max(outputs, 1)
    densenet_total += targets.size(0)
    densenet_correct += (predicted == targets).sum().item()

densenet_accuracy = densenet_correct / densenet_total
densenet_accuracies.append(densenet_accuracy)
```

9 Results

We now present our results for both OrthAdam and AttentionSplit. We will comment briefly on our interpretation of these results and how they line up with our expectations.

9.1 OrthAdam

9.1.1 CIFAR-10

As can be seen in chapter [8.1.1] in table [1] and figure [4], OrthAdam performs better than AdamW across all three neural network architectures. When comparing OrthAdam to AdamP, we see improvements across Densenet121 and ResNext50_32x4d, while Resnet18 performs better with AdamP.

9.1.2 CIFAR-100

Looking at the CIFAR-100 result, we observe a gain in performance across all three architectures for OrthAdam compared to it's parent algorithms. We observe that the gain from AdamW to AdamP is still more significant as compared to the gain from AdamP to OrthAdam, highlighting the overall importance of the novelty of [40].

Optimizer	Model	Accuracy (%)
AdamP	Resnet18	57.82
AdamP	Densenet121	62.94
AdamP	Resnext50_32x4d	59.19
AdamW	Resnet18	54.04
AdamW	Densenet121	57.56
AdamW	Resnext50_32x4d	51.43
OrthAdam	Resnet18	59.94
OrthAdam	Densenet121	64.41
OrthAdam	Resnext50_32x4d	61.21

Table 2: CIFAR-100 Results

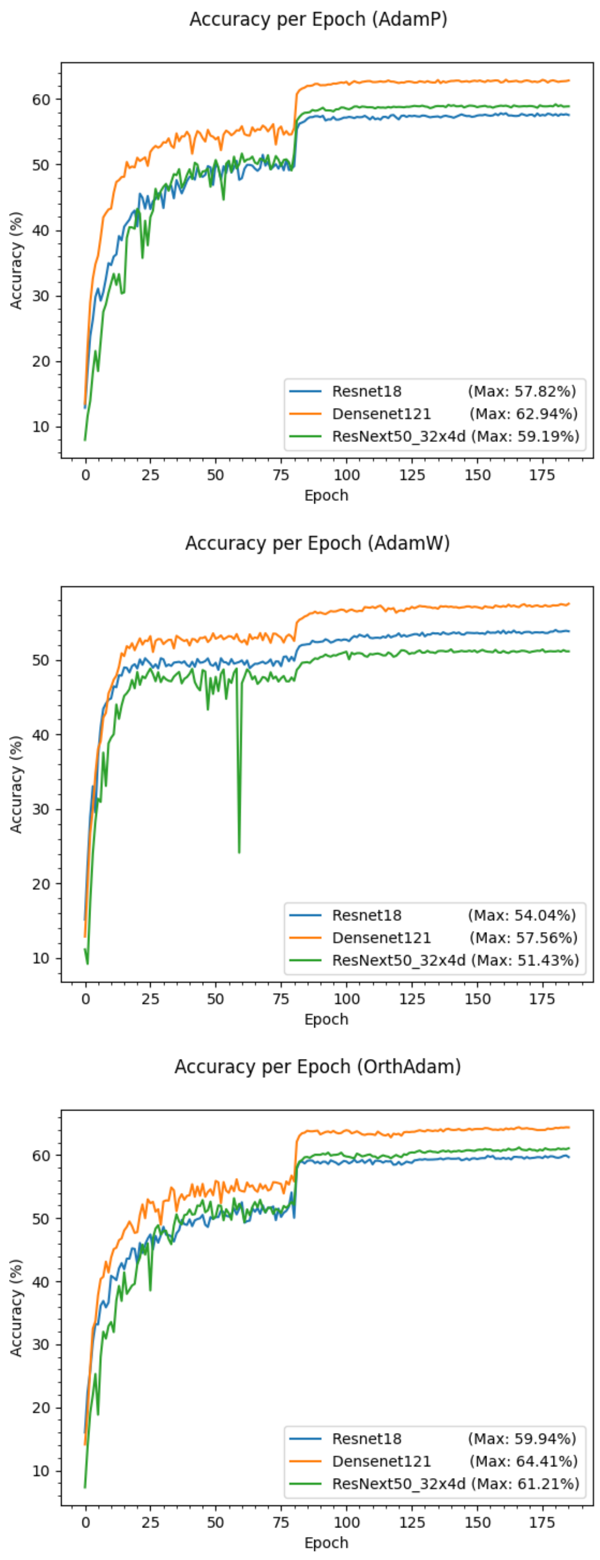

Figure 5: Accuracy Across Optimizers on CIFAR100

9.1.3 FMNIST

The results for FMNIST represents the most diverse of our results. Here we observe that all three optimizers have a single architecture for which it performs the best. These results represent an inconclusive finding among the other testing we have performed. We argue that this highlights the need to further research this line of optimizers.

Model & Environment	Accuracy (%)
FMNIST - Densenet121	92.97
FMNIST - Resnet18	93.23
FMNIST - Resnext50_32x4d	93.13

Table 3: Results from OrthAdam on FMNIST

9.1.4 SVHN: Street View House Numbers

The results for SVHN, as seen on figures [7] are more in-line with those observed for CIFAR-100 and CIFAR-10. We see in this case that AdamP is the overall leader for Resnet18 accuracy with remaining top accuracies lying with OrthAdam. In this case using the AdamP family of algorithms present a gain in performance across all three architectures over the parent algorithms.

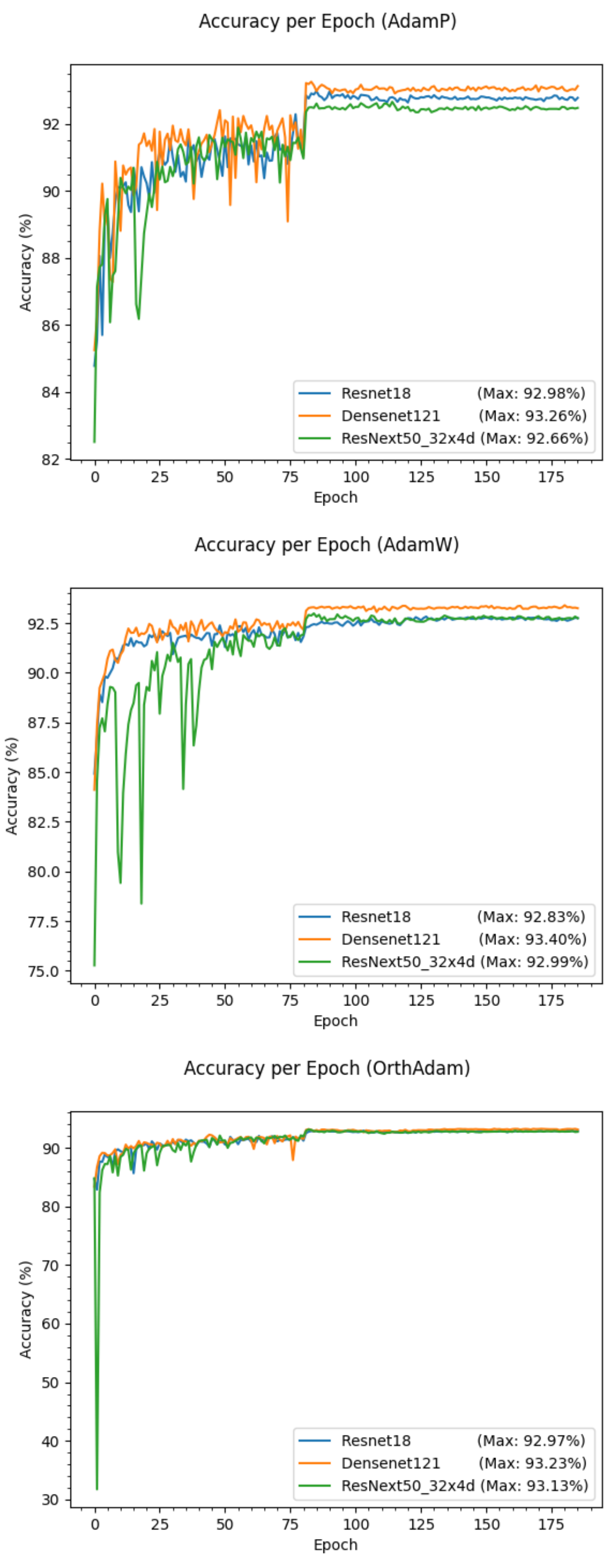

Figure 6: Accuracy Across Optimizers on FMNIST

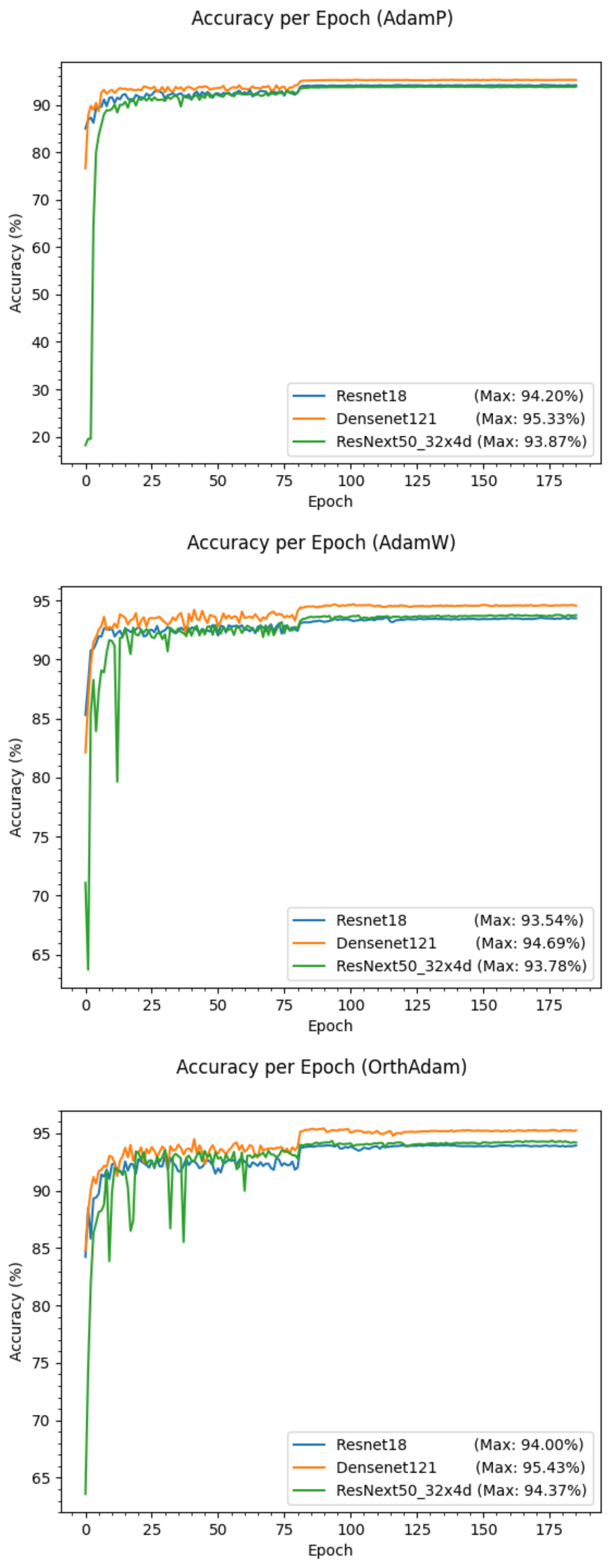

Figure 7: Accuracy Across Optimizers on SVHN

9.1.5 Summary

We observe for all our datasets improvements in total accuracy for at least 2/3 architectures when tested against the overall parent AdamW algorithm and immediate parent AdamP, with the exception being found in FMNIST, where each algorithm performed the best for one architecture. In most cases, we observed AdamP being the overall leader for Resnet-18 results, while for Densenet121 and ResNext50_32x4d OrthAdam obtains leading performance.

We concur heavily with the sentiment of [40], in the fact that the effects of scale-invarience and momentum based algorithms. We see the results of OrthAdam as but yet another step in the direction outlined in the parent algorithm AdamP.

9.2 AttentionSplit

9.2.1 OpenAI Gymnasium - Classic Control Environments

The following figures show the results we have from the setup as described by [7.4]. We observe leading mean accuracy in the test data for AttentionSplit in the Acrobot and CartPole environments, with LSTM being the overall leader for accuracy with MountainCar predictions. We believe these findings show the potential of combining Attention mechanism and recurrence mechanism for time-series prediction, with AttentionSplit serving as a catalyst in this case. We make no claim from this that AttentionSplit is conclusively better as compared to either LSTM or Transformer-encoder architectures, but merely attempt to show that there validity in the further study of the combination of these concepts for the set of problems encapsulated by this test-case. The results of this evaluation can also be found in table [4]

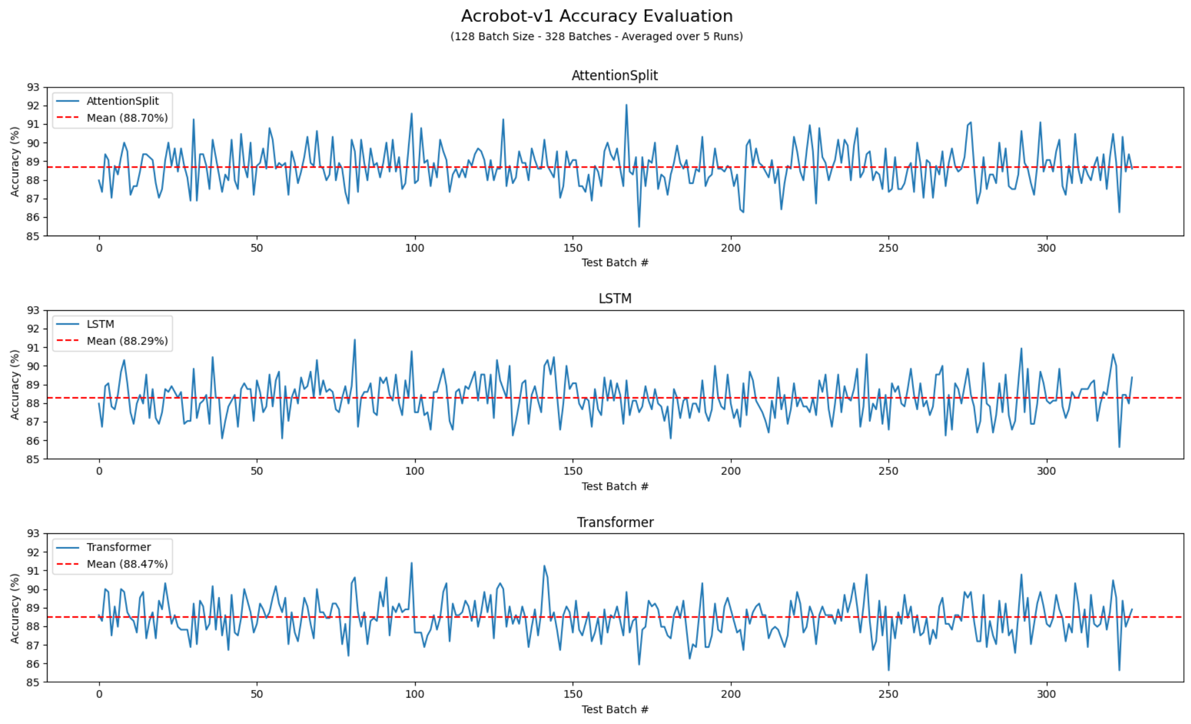

Figure 8: Accuracy on Time Series Prediction of Deep Q-Learning Agent on Acrobot-v1

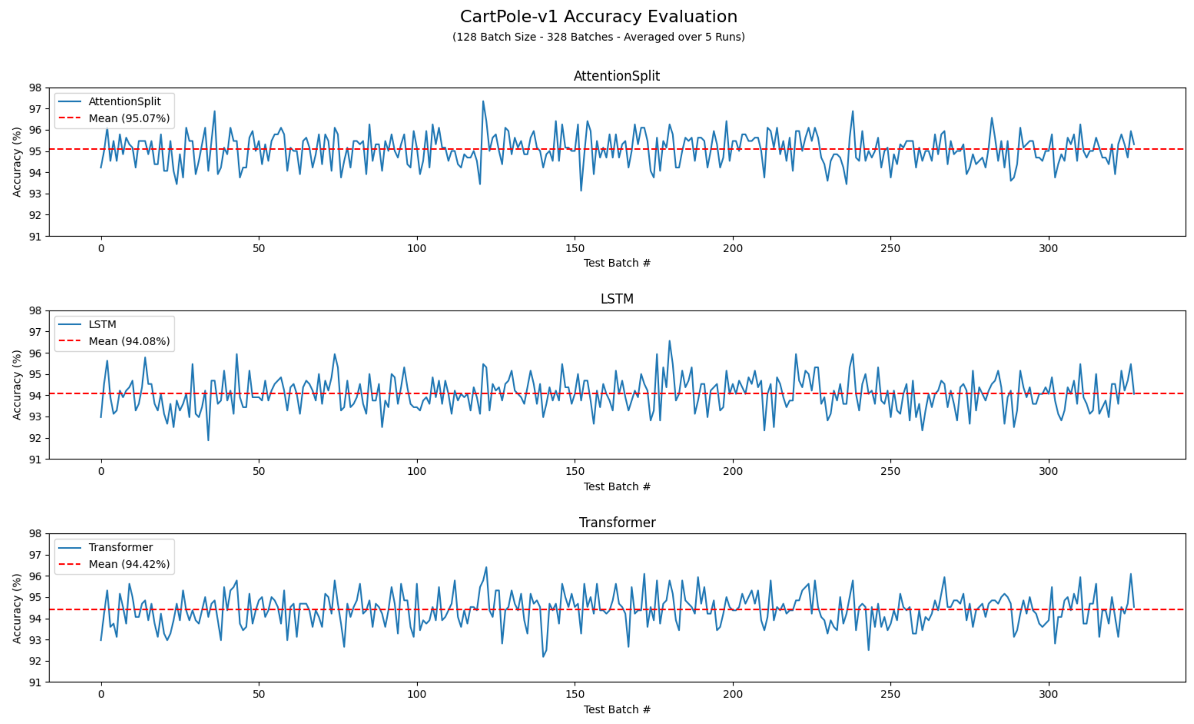

Figure 9: Accuracy on Time Series Prediction of Deep Q-Learning Agent on CartPole-v1

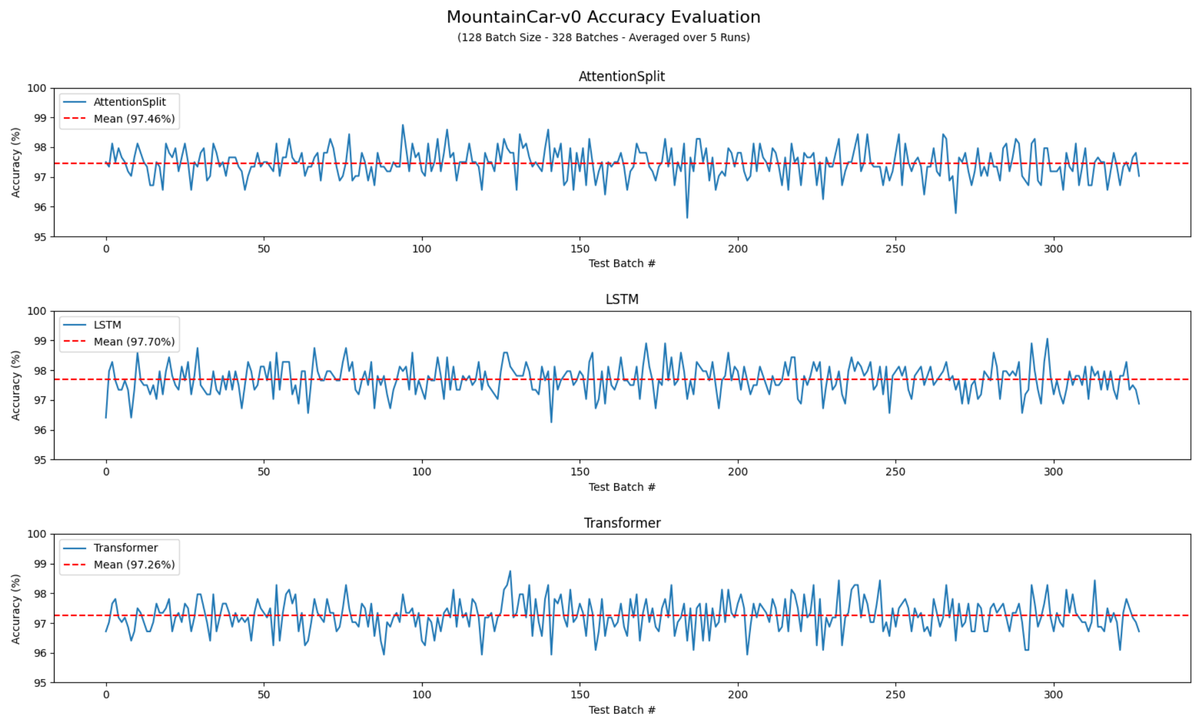

Figure 10: Accuracy on Time Series Prediction of Deep Q-Learning Agent on MountainCar-v0

Model & Environment	Mean Accuracy (%)
AttentionSplit - Acrobot-v1	88.70
AttentionSplit - CartPole-v1	95.07
AttentionSplit - MountainCar-v0	97.46
LSTM - Acrobot-v1	88.29
LSTM - CartPole-v1	94.08
LSTM - MountainCar-v0	97.70
Transformer - Acrobot-v1	88.47
Transformer - CartPole-v1	94.42
Transformer - MountainCar-v0	97.26

Table 4: Time-Series Evaluations from OpenAI Gymnasium Control Environments Deep Q-Learning Predictions

9.2.2 OpenAI Mujoco environments

9.2.2.1 HalfCheetah For half-cheetah, we observe both agents improving steadily, with a performance increase observed over the non-attention implementation of the agent, which can be seen on figure [11].

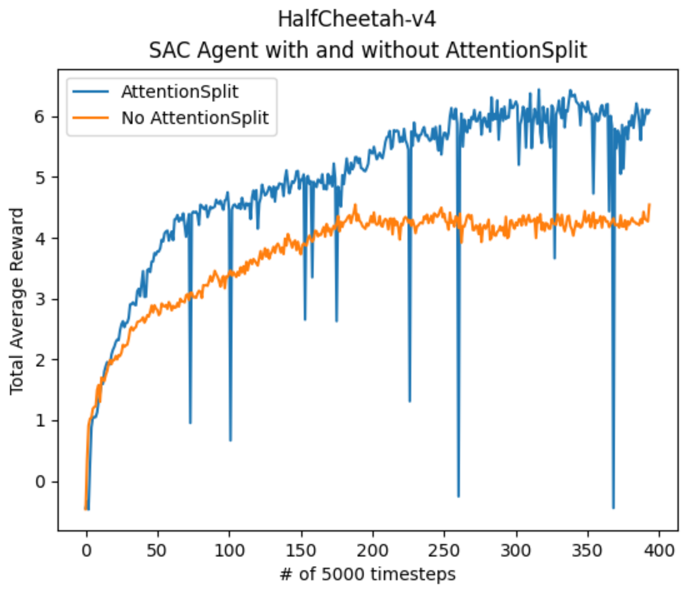

Figure 11: Average Rewards on HalfCheetah using a SAC agent with and without AttentionSplit

9.2.2.2 Walker2D In this case, we observe a somewhat steady learning pattern in the non-attention agent, with a total collapse of learning observed in the Attention agent. This presents an interesting finding as compared to the Half-cheetah environment results.

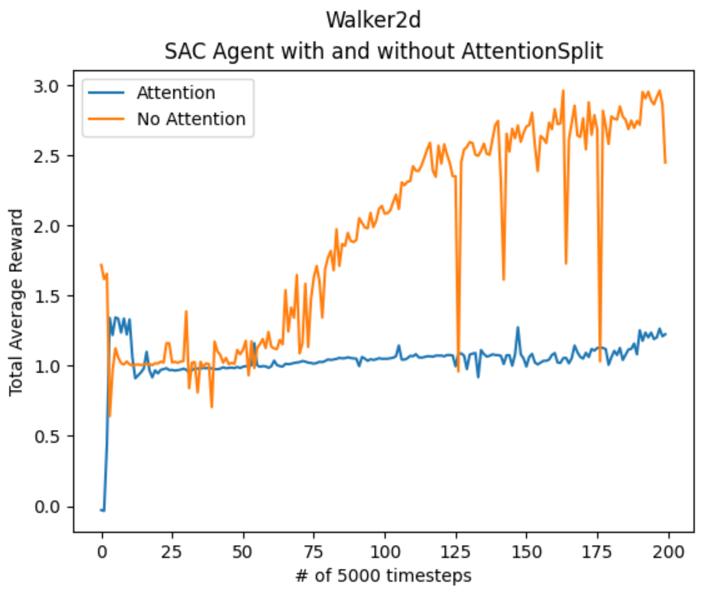

Figure 12: Average Rewards on Walker2D using a SAC agent with and without AttentionSplit

9.2.3 Summary

We observe that the combination of attention and recurrence can lead to performance gains in a selection of tasks, mostly encapsulated by time-series predictions in the control environments. We observe inconclusive results from the Mujoco environments, which we believe highlights the point of this work, which is to show that further study of these concepts in greater detail is necessary.

9.3 Implications

The practical implications of OrthAdam are given from the indicative results of maximal accuracy. Consider a case of a diagnostics system that has been trained using a gradient based optimizer. Here in these cases, improvements in the range of $1 - 2\%$ could be potentially life-altering, and even ranges of percentages lower than whole numbers can be important for providing the best care solutions.

In the case of AttentionSplit, we will conclude that the main takeaway as we see it, is the highlight of the combination of attention mechanism with recurrence. There is a time and a place for this concept in the area of temporal data analysis, which we believe is illustrated through our very simple model proposal and given by the results of our testing. While AttentionSplit has less certain associated "success" as module compared to OrthAdam, we believe this is simply in line with our mission, which as simply to show the potential strengths of a combined approach.

10 Future Improvement

We have many ideas of how to further explore the ideas dealt with in this work. One main limiting factor that we have not talked about is a of a practical nature. All testing done in this work was performed on the personal computers available of us. This has been a bottleneck on same aspects of this project, as simply testing becomes a multi-day ordeal and bug-fixing that become apparent at run-time become costly with regards to time. We believe the readers will find much benefit in securing better computational resources in the pursuit of furthering ML studies.
First - for OrthAdam and AttentionSplit - we suggest more comprehensive testing to be conducted. In the case of OrthAdam, we performed almost none of the tests associated with it's parent algorithm AdamP, such as MS COCO. These tests are important bench-marks for Computer Vision tasks and would either further validate the findings of our work, or help to put them into question. Again in the case of OrthAdam, another area that would be a particular worth would be a more comprehensive mathematical analysis of the implications of the scale-invariant assertion from [31] on lemma 2.2 from [40].

For AttentionSplit, further testing in more diverse areas of temporal data analysis would be of great interest, such as machine translation tasks, implementations similar to [45] for the Atari environment, but with the LSTM component exchanged for a AttentionSplit or similar attention-recurrence mechanism is a test that could serve as a good baseline of comparison with other modules such as LSTM. We also want to emphasize the simplicity of the current attention-recurrence mechanism of AttentionSplit. More sophisticated combinations - such as formally introducing gating mechanism to the recurrent part and using multi-head scaled dot product attention for the attention part - could potentially significantly boost the performance of attention-recurrence compared to the AttentionSplit implementation.

11 Final Remarks

Through this work we show the importance of further research in two key areas of deep learning. We have designed and implemented an Adam optimizer variant named *OrthAdam* that shows baseline improvements in maximal accuracy across a selection of bench-marks. Similarly, we designed and implemented an attention-recurrence mechanism that we believe shows the potential of combining the ideas of attention and recurrence in specially designed modules.

We conclude that the idea combining attention mechanisms with recurrence mechanisms - as was the original goal of the thesis - has been shown to be useful and therefore of importance to the literature.

We hope this work serves as a basis of interest for the readers to consider the ideas put forth, but not take the current implementations as ”final” solutions to these areas.

References

[1] Jason Wang, Luis Perez - *The Effectiveness of Data Augmentation in Image Classification using Deep Learning*, Stanford University.

[2] Connor Shorten, Taghi M. Khoshgoftaar - *A survey on Image Data Augmentation for Deep Learning*, Journal of Big Data, 2019

[3] V.V. Berezovsky1, N.V. Vygovskaya - *An overview of neural networks for medical image recognition*, E3S Web of Conferences 460, 04028 (2023)

[4] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, C. V. - *JawaharThe Oxford-IIIT Pet Dataset*

[5] Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, Yi Yang - *Random Erasing Data Augmentation*, Cognitive Science Department, Xiamen University, China §University of Technology Sydney

[6] Luis P, Jason W. The effectiveness of data augmentation in image classification using deep learning. In: Stanford University research report, 2017.

[7] Ian J. Goodfellow, Jean Pouget-Abadie†, Mehdi Mirza, Bing Xu, David Warde-Farley,Sherjil Ozair‡, Aaron Courville, Yoshua Bengio - *Generative Adversarial Networks*, Departement d’informatique et de recherche op ´ erationnelle, Universite de Montreal, Montreal, QC H3C 3J7

[8] Gatys, Leon A.; Ecker, Alexander S.; Bethge, Matthias (26 August 2015) - *A Neural Algorithm of Artistic Style*

[9] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, Jiawei Han - *On the Variance of the Adaptive Learning Rate and Beyond*

[10] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby- *Big Transfer (BiT): General Visual Representation Learning*, Google Research, Brain Team Z¨urich, Switzerland

[11] Ross Wightman, Hugo Touvron, Herve Jegou - *ResNet strikes back: An improved training*,

[12] Masanari Kimura - *Understanding Test-Time Augmentation*

[13] OpenAI's Website as of the 13th of May 2024.

[14] Richard S. Sutton - *Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding*

[15] OpenAI Gymnasium - Acrobot Environment as of the 13th of May 2024.

[16] P. Wawrzyński - *A Cat-Like Robot Real-Time Learning to Run*

[17] OpenAI Gymnasium - HalfCheetah Environment as of the 13th of May 2024.

[18] OpenAI Gymnasium - Walker2D Environment as of the 13th of May 2024.

[19] CIFAR-10 Dataset

[20] FMNIST Dataset

[21] SVHN: Street View House Numbers Dataset

[22] Book 3 of Openstax Calculus series

[23] James Martens, Roger Grosse - *Optimizing Neural Networks with Kronecker-factored Approximate Curvature*, Department of Computer Science - University of Toronto

[24] Yeming Wen*, Kevin Luk*, Maxime Gazeau*, Guodong Zhang, Harris Chan, Jimmy Ba - *Stochastic Gradient Descent with Covariance Noise*

[25] Stanislaw Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, Amos Storkey - *Three Factors Influencing Minima in SGD*

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller - *Playing Atari with Deep Reinforcement Learning*

[27] Kaleigh Clary, Emma Tosch, John Foley, David Jensen - *Let's Play Again: Variability of Deep Reinforcement Learning Agents in Atari Environments*

[28] OpenAI Gymnasium - Atari Space Invaders as of the 13th of May 2024

[29] PyTorch

[30] Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton - *On the importance of initialization and momentum in deep learning*

[31] Diederik P. Kingma*, Jimmy Lei Ba - *Adam: A Method for Stochastic Optimization*

[32] Ilya Loshchilov, Frank Hutter - *Decoupled Weight Decay Regularization*

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser*, Illia Polosukhin - *Attention Is All You Need*

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun - *Deep Residual Learning for Image Recognition*

[35] Richard S. Sutton (Author), Andrew G. Bart - *Adaptive Computation & Machine Learning*

[36] Dan Hendrycks, Kevin Gimpel - *GAUSSIAN ERROR LINEAR UNITS (GELUS)*

[37] Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton - *Layer Normalization*

[38] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov - *Improving neural networks by preventing co-adaptation of feature detectors*

[39] Günter Klambauer, Thomas Unterthiner, Andreas Mayr - *Self-Normalizing Neural Networks*

[40] Byeongho Heo, Sanghyuk Chun, Seong Joon Oh, Dongyoon Han, Sangdoo Yun, Gyuwan Kim, Youngjung Uh, Jung-Woo Ha - *Adamp: Slowing down the Slowdown for Momentum Optimizers on Scale-invariant Weights*

[41] Tom Erez, Yuval Tassa†, Emanuel Todorov - *Infinite-Horizon Model Predictive Control for Periodic Tasks with Contacts*

[42] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine - *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller - *Playing Atari with Deep Reinforcement Learning*

[44] Sepp Hochreiter, Jurgen Schmidhuber - *Long Short-Term Memory*

[45] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, Will Dabney - *Recurrent Experience Replay In Distributed Reinforcement Learning*