MINISTRY OF EDUCATION
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

PHAN NGỌC LÂN

GRADUATION THESIS

BK.Synapse: Distributed Deep Neural Network Training for Object Detection

Instructor: Dr. ĐINH VIẾT SANG

Hanoi, 5 - 2019

Acknowledgments

Abstract

Training deep neural networks efficiently has been a thoroughly-researched topic over the history of deep learning. However, training at scale usually means adding on multiple layers of complex deployment logic and parallelization concerns, distracting researchers from the core of their algorithms. This thesis presents a framework called BK.Synapse that can facilitate distributed training while maintaining clarity, simplicity, and user-friendliness. The design is modular, allowing flexible and easy deployment on a variety of hardware specifications. We benchmark BK.Synapse in a case study: training a neural network for the object detection problem. Our results show a good amount of speedup over conventional training, with very few modifications to the existing codebase.

Contents

List of Figures

List of Tables

Chapter 1

Problem Statement

1.1 Training Neural Networks at Scale

Deep neural networks (DNNs) have seen drastic progress within the past several years. Improvements in network architecture and hardware capabilities have resulted in state-of-the-art performance in many cognitive tasks, notably in computer vision, natural language processing and audio processing.

Training neural networks is a very computationally intensive task, which can take days or even weeks to complete. This makes experimenting and optimizing them very difficult. However, as DNNs process training examples in batches, they can be very scalable. Therefore, a framework that provides the capacity for large scale training can greatly improve training speed, and consequently network quality.

At the same time, distributed environments make it difficult to validate and test new networks. A preferable workflow would be testing a network locally, and deploying to multiple nodes later. As such, minimizing the amount of work required to port between these workflows is also a priority. This thesis aims to design and implement a system that satisfies these criterias.

1.2 Object Detection

Object detection is a computer vision problem that aims to detect instances of semantic objects of a certain class (such as humans, buildings, cars, etc...) in digital images and videos. Formally, object detection is often defined as: given an input image, produce a correct set of bounding boxes and corresponding labels for each defined object within

the image. An example is shown in Figure 1.1.

Figure 1.1: An example of object detection[1]

Object detection has a wide range of applications, most notably in surveillance (detecting people, movement,...) and image retrieval (using detected objects as image tags). It is also commonly applied in conjunction with other computer vision tasks, such as classification, eg. an object of a generic class may be detected and then further classified.

The more difficult task of finding the exact bound for objects (as opposed to bounding boxes) is called semantic segmentation.

1.2.1 Detecting Document Text Regions

Physical documents, such as books, reports, receipts,... still hold a large amount of information that's often inaccessible to computers. Digitizing these documents can open up a lot of possible applications, such as archiving historic texts, automatically grading exams, or managing personal finance.

An important task in digitizing these documents is finding the text regions. For documents with diverse structures (such as receipts or flyers), this is not trivial.

This thesis approaches the text region detection problem as an object detection problem. We use this problem as a case study to showcase the use of our framework.

[1] https://hackernoon.com/how-visual-object-detection-can-transform-manufacturing-industries-8b6698cc0a47

Chapter 2

Theoretical Overview & Related Works

2.1 Machine Learning

Machine Learning (ML) is the study of algorithms and statistical models that perform tasks without explicit instructions, but by learning and inferring from data. Machine learning algorithms produce "models" from data during their training phase, and infer the model's outputs during runtime to produce results on unseen (test) data.

Machine learning is a subfield within artificial intelligence. Most machine learning problems attempt to solve human-centric tasks, such as visual cognition, or language understanding, etc,... Since machine learning is approximate by nature, problems that would require machine learning solutions are often NP or incomputable.

2.1.1 Relation to statistics

Machine learning is closely related to statistics. A machine learning model provides prediction based on a statistical model of its training data. Although they aim to generalize to new examples, these models are still heavily bound to the training set's distribution. Understanding the statistical properties for each problem is an important step in creating accurate models.

2.1.2 Relation to optimization

Machine learning also has intimate ties to optimization: many learning problems are formulated as minimization of a loss function on the training set. The loss function represents the discrepancy between the model's predictions and the actual problem

instances.

The key difference between machine learning and the above fields is their goals. Statistics and optimization both aim to extract results from the given (training) data, whereas machine learning is concerned with generalizing to unseen data.

2.1.3 Basic concepts

Types of learning

There are several types of learning algorithms, including: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning.

Supervised algorithms learn from a dataset containing both the inputs and desired outputs (labels). The trained model contains a function that associates any given input with an output, and aims to produce correct outputs for both seen and unseen data. Two sub-types of supervised learning are regression (where the output is a continuous value) and classification (where the output is one or several discrete classes). Common supervised learning algorithms include Naive Bayes classification, Support Vector Machines (SVMs), Decision Trees and Neural Networks. This is the most common type of machine learning, used in domains such as image classification, sentiment analysis, or speech recognition, etc,...

Semi-supervised algorithms are similar, however they are designed to handle missing labels or very small datasets, often by making heuristic assumptions for the problem.

Unsupervised algorithms, on the other hand, make use of unlabeled data, where only the input is known. Often, the goal for these models is to learn a relationship between the examples. Instead of responding to feedback, unsupervised learning algorithms identify commonalities in the data and react based on the presence or absence of such commonalities in each new piece of data. Popular unsupervised methods include k-means Clustering, and Neural Autoencoders. A major application for unsupervised learning is cluster analysis, in which entities (such as users, products, etc,...) can be classified into groups based on their features.

Finally, reinforcement learning algorithms are modeled as agents within an environment. The environment provides feedback to the agent's actions, which are interpreted as reward values. The goal in reinforcement learning is to learn a policy for the agent to perform which optimizes its reward, and ultimately achieve its task. Notable applications of reinforcement learning include self-driving cars and game-playing AIs (eg.

Google's AlphaGo [28]).

Partitioning datasets

Evaluating the effectiveness of ML models differ from other algorithms and statistical models. In this section, we focus on evaluating supervised models.

Supervised ML models are trained on a training dataset, however the goal is to produce accurate predictions on unseen data outside this training set. Hence, it's impossible to evaluate on training data, otherwise a model that "learns" by simply memorizing the training set would score perfectly, yet unusable. As such, a subset of labeled data is set aside from training as the test dataset, which only serves to evaluate the model.

Additionally, many models have hyperparameters that require selection, often by experiment. This selection should also be done on unseen data, yet it's unfair to use the test set as this simply picks out whichever parameter set that happens to fit the test distribution. Instead, a seperate validation dataset is held out for this purpose.

For large datasets, a straightforward split of 3 datasets is often made with a ratio of around 70/15/15 for training, validation and testing. In smaller datasets, an alternate approach is k-fold cross validation, where the entire dataset is split into k equal parts (folds). Training is then performed k times, each using one fold as the validation and test set, and the rest as the training set. Evaluation results are then averaged among all runs.

When splitting datasets, it can sometimes be important to maintain the distribution of classes or other feature properties in the original data. This technique is called stratified sampling, and is especially critical for datasets that are unevenly distributed.

Evaluation metrics

Different metrics are used to evaluate machine learning models, depending on the problem domain. Most often, metrics focus on how accurate the model is in performing the given task, however can also include other requirements like efficiency, robustness, scalability or interpretability.

For classification models, the most common metric is accuracy, which measures the ratio of correct classifications made by the model. However, this metric can be noisy on unbalanced datasets. Specifically, if 90% of the dataset belongs to class A, then a classifier that always predicts class A would have 90% accuracy (!). Therefore, 3 other

metrics are usually added to evaluation: precision, recall, and F1 score.

Precision measures the ratio between true positives (number of accurate predictions made by the model for one class) to all predictions in that class. Recall measures the ratio between true positives for a class and all examples labeled as that class. Finally, F1 score is the harmonic mean between precision and recall. Calculating these metrics is often done using a confusion matrix (Figure 2.1).

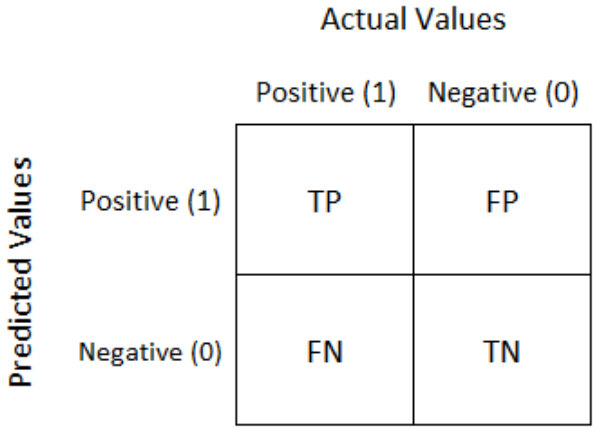

Figure 2.1: A confusion matrix
(TP: true positive, FP: false positive, FN: false negative, TN: true negative)[1]

More specifically, these metrics are calculated as follows:

$$Precision_i = \frac{TP_i}{TP_i + FP_i} \tag{2.1}$$

$$Recall_i = \frac{TP_i}{TP_i + FN_i} \tag{2.2}$$

$$F_1^i = 2\frac{Precision_i \cdot Recall_i}{Precision_i + Recall_i} \tag{2.3}$$

In essence, precision penalizes a model when making false predictions of the given class, while recall penalizes a model's inability to cover all examples of that class. A model that tries to "cheat" in one metric or class (like how our hypothetical "always A" model would have perfect recall for class A) invariably lower other metrics (its precision and recall for classes that aren't A would be zero), thus still having a low F1 score.

[1]https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

2.2 Neural Networks & Deep Learning

2.2.1 Artificial Neural Networks

Artificial neural networks (commonly referred to as neural networks) are a class of machine learning models inspired by the way biological neural systems process data.

A biological neural network is composed of a group or groups of chemically connected or functionally associated neurons. A single neuron may be connected to many other neurons and the total number of neurons and connections in a network may be extensive. Connections, called synapses, are usually formed from axons to dendrites, though dendrodendritic synapses and other connections are possible.

Artificial neural networks employ a much more simplified approximation of this process, where neurons are replaced with simple computational units that perform transformation operations on input data. Despite each neuron's simplicity, the amount of neurons along with their connections allow neural networks to represent incredibly complex functions.

The first neural network was proposed in 1958 by Frank Rosenblatt [25], called the Perceptron. A perceptron is basically a single neuron, which consists of a set of weights $W = (w_0, ..., w_n)$ and an activation function $f(x)$ (in this case the unit step function). With a continuous vector $X = (1, x_1, ..., x_n)$ as input, the output of a perceptron is represented by the function:

$$y = f(W \cdot X) \tag{2.4}$$

The idea of the perceptron eventually expanded into multi layer perceptrons, or better known as artificial neural networks. Architecturally, they differ in that there are multiple "neurons" (or computational units) as opposed to one, and they are divided into layers (Figure 2.2).

Three types of layer are present in a neural net: the input layer, hidden layers, and the output layer. The input layer is the representation of input data, where each unit simply contains the feature value of an example. Thus, this layer is bound by the size of our feature representation. The hidden layers are transformations performed by the network on the input, each working similarly to a perceptron. Finally, the output layer contains the output values of the network. Typically, this layer returns the values we wish to use as the final result, or values that can be inferred to produce it.

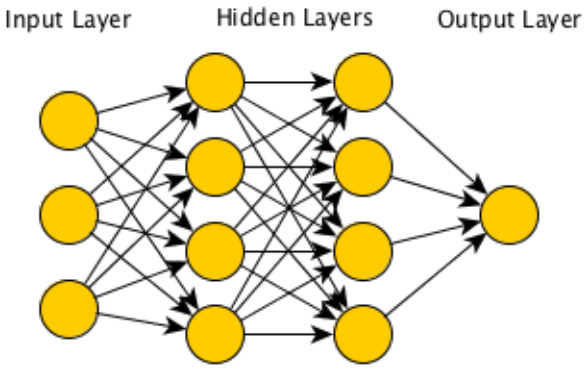

Figure 2.2: Architecture of a 4-layer neural network[2]

In this classic architecture, each neuron in layer i is connected to every neuron in layer $i+1$. As such, these networks are also referred to as fully connected networks.

Neural networks implement 2 primary procedures: forward pass and backpropagation.

Forward pass

The forward pass is used to get output results from a neural network. Starting with an input example $X = (x_1, ..., x_n)$ at the input layer, we iterate through subsequent layers one by one, calculating the output at each pass (Algorithm 1). The output for layer i is defined by:

$$y_i = f_i(y_{i-1} \cdot W_i) \tag{2.5}$$

where f_i is the layer's activation function, and W_i is the layer's weight matrix. The size of W_i corresponds to the number of neurons at layers i and $i-1$.

In order for neural networks to represent complex, non-linear models, the activation function at each layer is often non-linear. The most widely used functions are *sigmoid* and *tanh* (Figure 2.3).

Backpropagation

Neural networks learn from examples using backpropagation. As the name implies, backpropagation works by tracing back from the output layer. Given a training example

[2]https://technology.condenast.com/story/a-neural-network-primer
[3]https://www.researchgate.net/figure/The-sigmoid-and-hyperbolic-tangent-activation-functions_fig2_2654867842

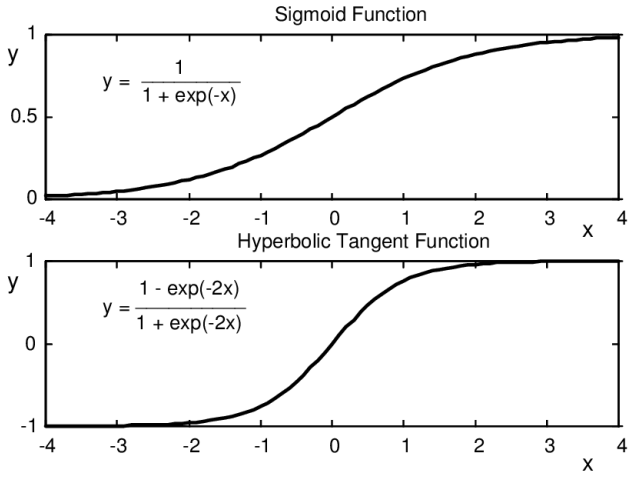

Figure 2.3: Sigmoid and tanh activation functions[3]

Algorithm 1: Forward pass

Input : $X = (x_1, ..., x_k)$: a set of input features
$L = ((W_1, f_1), ..., (W_m, f_m))$: the network's layers
Output : y_m: the network's prediction $Z = (z_1, ..., z_m)$: the output at each layer

$y_0 \leftarrow X$
for $i \in (1...m)$ **do**
  $z_i \leftarrow y_{i-1} \cdot W_i$
  $y_i \leftarrow f_i(z_i)$
end
return y_m, Z

$X = (x_1, ..., x_n)$ and the label Y, we first get the network's prediction y using the forward pass. The general idea is to "propagate" the error signal between y and Y to the whole network, or in other words, update the network to decrease prediction error.

The error signal is calculated using a loss function. Different problems may utilize different loss functions. Typically, regression problems employ Mean Squared Error (Equation 2.6), while classification problems use the Cross Entropy loss (Equation 2.7). Other notable functions include Triplet Loss as seen with FaceNet [26], and Adversarial Loss used by Generative Adversarial Networks [8].

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - Y - i)^2 \tag{2.6}$$

where n is the total number of examples.

$$H = -\frac{1}{n}\sum_{i=1}^{n}(Y_i log(y_i) + (1 - Y_i) log(1 - y_i)) \tag{2.7}$$

where n is the total number of examples.

In order to update the network, we need to know the values with which to update each neuron. Backpropagation finds these values for each layer i using the gradient w.r.t layer $i + 1$. We start by calculating the gradient at the output layer m w.r.t the loss function. Then, using the derivative chain rule, calculate the gradient at each previous layer. This is described in detail in Algorithm 2.

Algorithm 2: Backpropagation

Input : $X = (x_1, ..., x_k)$: a set of input features
Y: the correct example label
$L = ((W_1, f_1), ..., (W_m, f_m))$: the network's layers
Output : $\Delta = (\delta_1, ..., \delta_m)$: the gradient at each layer

$y, Z \leftarrow forwardPass(X)$
$C \leftarrow loss(y, Y)$
$\delta_m = \frac{\delta C}{\delta w_m} \odot f'_m(z_m)$
for $i \in (m - 1...1)$ **do**
  $\delta_i \leftarrow (w_{i+1}^T \cdot \delta_{i+1}) \odot f'_i(z_i)$
end
return Δ

Gradient descent

With the gradient at each layer, we can update the network by "moving" in the opposite direction of the gradient vectors. Given a small enough step, we can guarantee that the loss function decreases. By repeating this procedure on the training dataset, the network can converge at a minimum on the loss function. This procedure is called gradient descent (Algorithm 3).

Algorithm 3: Gradient Descent

Input : $X = (x_1, ..., x_k)$: a set of input features
Y: the correct example label
$L = ((W_1, f_1), ..., (W_m, f_m))$: the network's layers
γ: the learning rate

Output : The updated network

repeat
  $\Delta \leftarrow backprop(X, Y)$
  $W \leftarrow W - \gamma\Delta$
until *convergence*;

A learning rate is specified to adjust the step for each gradient descent iteration. A high learning rate means the network is updated with large strides, which makes it susceptible to divergence at later steps. A low learning rate is more guaranteed to converge, but takes more iterations. Conventionally, the learning rate is set in the range of 10^{-2} to 10^{-4}.

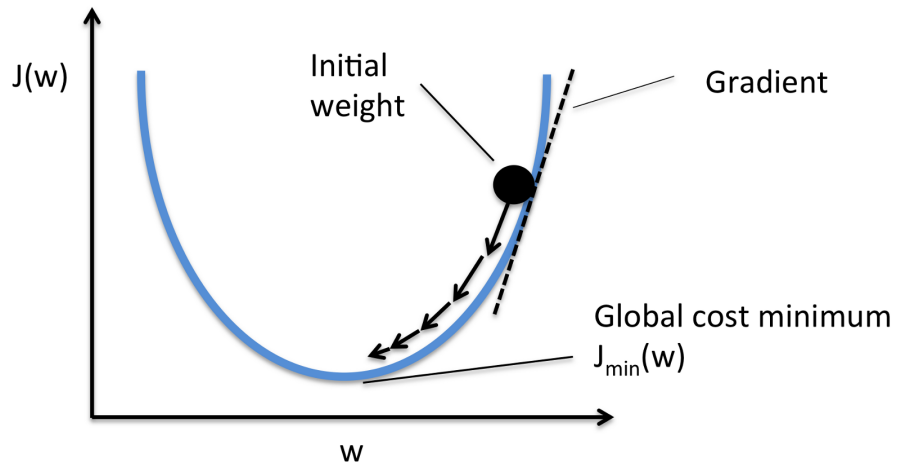

Figure 2.4: Gradient descent schema[4]

Gradient descent accounts for the gradient over the entire dataset. For very large datasets, this is not feasible. Minibatch gradient descent is a modified version that only updates the model on several examples at a time (one batch). This version runs in multiple "epochs", each of which goes through all batches in the dataset. Stochastic gradient descent (SGD) is a version with a batch size of 1. Stochastic and minibatch gradient descent usually requires lower learning rates and more iterations, but still consistently converges with lower memory footprints.

Additionally, a lot of research has been conducted on adapting the learning rate during the training process, most notably resulting in several optimizers such as Adam

[4]https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

[13], Adadelta [32] and RMSprop [30].

In general, networks with more layers and neurons can better fit the training data. However, simply increasing the number of neurons does not always result in more accurate models. Wide neural nets overfit very quickly, while networks deeper than 5-6 layers suffer from vanishing gradients. Because of these limitations, the traditional neural net architecture are only viable at a certain size.

Deep Learning techniques attempt to create larger and deeper networks while overcoming these challenges. These techniques include network architectures, optimizers, activation functions, etc,... In the next section, we look into Convolutional Neural Networks, one of the most impactful models in Deep Learning.

2.2.2 Convolutional Neural Networks

Convolutional Neural Networks (ConvNets, or CNNs) was first designed to work with images. Their architecture addresses a key problem when using ANNs to process images, as the input size can be very large (eg. 200 x 200 x 3), which quickly leads to huge, expensive networks that overfit the data.

A typical CNN is constructed from several components:

The INPUT layer is the same as a fully connected network, which holds the input features.

The CONV (convolution) layer is the core building block of a Convolutional Network (Figure 2.5). Its parameters consist of a set of learnable filters. Filters are small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height with 3 color channels). During the forward pass, we slide/convolve each filter across the width and height of the input volume and compute dot products between the filter and the input at any position. As the filter slides across the input volume, it produces a 2-dimensional activation map that representing its responses at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a color blotch, to specific, complex patterns on higher layers of the network. Activation maps from different filters are stacked along the depth dimension and produce the output volume.

A key difference between CONV layers and regular hidden layers is their local connec-

[5] http://cs231n.github.io/understanding-cnn/

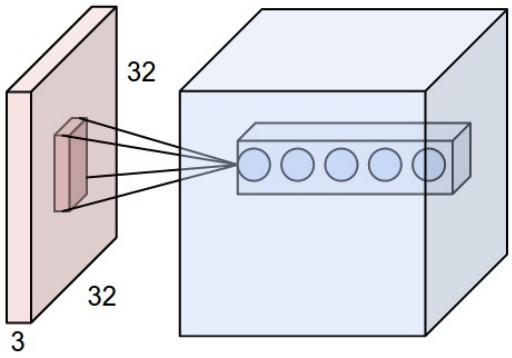

Figure 2.5: An example convolution layer[5]

tivity. Each neuron is connected to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field (or filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space, but always full along the entire depth of the input volume.

The assumption made by convolution is that if a feature is a good representation for one region of the image, then it's probably good for other regions as well. As the filters convolve, the weights used for each is the same, and their gradients accumulate during backpropagation. This allows CONV layers to share weights between input regions, drastically reducing their size while still covering the entire input.

In lieu of activation functions like *sigmoid* or *tanh*, CONV layers are usually activated using the ReLU function, with $ReLU(x) = max(x, 0)$. ReLU has several properties that helps with very deep networks:

- ReLU is a non-linear function
- The gradient for ReLU is either 1 or 0, meaning it's not affected by vanishing gradient

One downside for ReLU is the existance of "dead neurons" whose output is negative. These neurons would have zero gradient and can no longer learn or contribute to the network. Alternative functions such as Leaky ReLU or Exponential ReLU have been proposed to overcome this.

Next, POOL (pooling) layers are used to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The POOL layer operates independently on every depth

slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged (Figure 2.6).

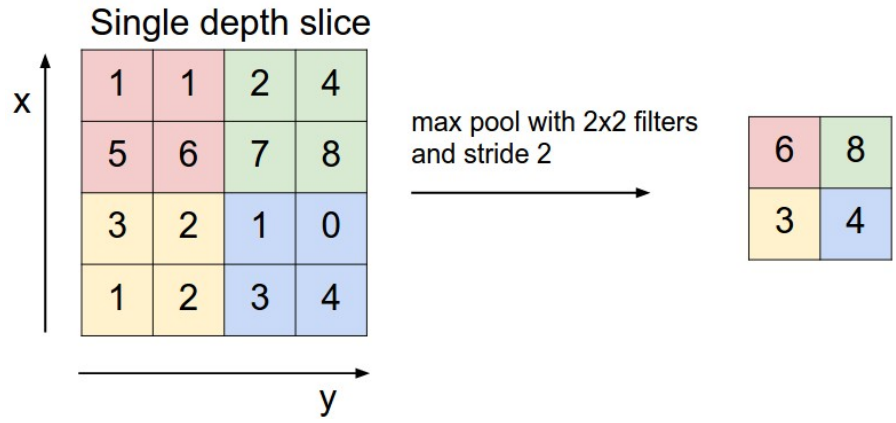

Figure 2.6: An example of max pooling[6]

In addition to max pooling, the pooling units can also perform other functions, such as average pooling or even L2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.

CNNs are constructed using "blocks" of (CONV, ReLU, POOL) layers of different sizes, and finally followed by several fully connected layers to produce output results if needed (eg. classification result). VGGNet [29] is a very effective network that follows this architecture.

Later architectures introduce more complex, non-linear ordering of convolution blocks to facilitate learning. ResNet [10], for example, introduces skip connections (forming what's called "residual blocks") between convolution blocks.

2.2.3 Object Detection With Convolutional Neural Networks

ConvNets have been successfully applied to many computer vision problems, the most common being classification in which application is quite straightforward (using fully connected layers as the output layer). For object detection, modeling the problem's inputs and outputs with CNNs is somewhat more complex. Two types of model have

[6]http://cs231n.github.io/understanding-cnn/

been proposed and extensively researched to varying results: two-stage detection and single-stage detection.

Two-stage detection splits the problem into two subtasks: finding regions of interest (RoI) where there may be objects, then locating and classifying the single object within each region. The tasks are split out to handle the varying number of objects within each image. Faster R-CNN [23] is a good example of a two-stage detector. The input image is put through convolution layers to produce the initial feature map. Then, a small ConvNet is used to generate the ROIs, called the Region Proposal Network. Because ROIs can have arbitrary sizes, they are reshaped using an ROI Pooling layer. ROI pooling works similarly to max pooling, but instead of the whole feature map, it takes the feature map segment corresponding to each ROI, divides it into $m \times n$ section and performs max pooling on those sections to produce the $m \times n$ output. In essence, ROI pooling squashes ROIs of different sizes into uniform representations of their strongest signals. Afterwards, each ROI map is put through a classifier feedforward network to generate the class label. The loss function for the network is the sum of the classification loss for each correctly predicted region, and the regression loss for each predicted box.

In contrast, one-stage detectors model the problem as classifying each subregion on the image as either one of the pre-defined class (foreground) or no class at all (background). YOLO [22] and SSD [18] are both good examples of one-stage detectors. Both architectures define multiple overlapping boxes on the feature map region. The network then learns how to correctly classify each of these boxes into the correct class, as well as the offset to the default position. Overlapping boxes of the same class are combined into a single bounding box using a procedure called Non-Maximum Suppression.

Non-Maximum Suppression (NMS) is a procedure that takes a series of overlapping bouding boxes as input, and outputs the pruned list of non-overlapping boxes. NMS achieves this by prioritizing boxes with the lowest bottom-right corner, and removing any overlapping box over a predefined threshold.

For both types of detector, the model needs to pinpoint the coordinates for each bounding box. Although networks like Faster R-CNN can accomplish this by simply using the offset from a $(0, 0)$ root, this creates bias for larger boxes. Instead, fixed anchor boxes are used as base for the offsets. To best reduce bias, the anchors' shapes and ratios should cover the range of desired image shapes.

Several metrics are introduced to measure object detectors' performance. Intersection-over-Union (IoU) is commonly used to measure box location accuracy. Given a predicted bounding box and a ground truth box, IoU is computed as the ratio between their in-

tersection area and their union area:

$$IoU = \frac{S_{overlap}}{S_{union}} \tag{2.8}$$

A higher IoU means the box prediction more closely matches the ground truth box.

Average precision (AP) is the most common and standard metric for evaluating object detectors. Average precision is calculated by first plotting the precision-recall curve. Given a set of positive prediction of the class, sort them by confidence level. As we accumlate through the sorted list, precision fluctuates according to whether the prediction is correct, while recall increases at each correct prediction, creating a zig-zag pattern. Average precision is calculated as the area-under-the-curve of this precision-recall curve.

For object detection, AP is calculated with regards to IoU, in which predictions that overlaps a ground truth box above a certain IoU are considered positives.

2.3 Computational Graphs & Deep Learning Frameworks

2.3.1 Computational Graphs

Although they are commonly visualized as layers of neurons, neural networks are more often represented as computational graphs.

Computational graphs are directed graphs that express a series of computations. Each node is either an input variable, or, when there are incoming edges, a function of those edges' tail nodes. Edges simply denote dependencies between nodes (eg. $a = b + c$).

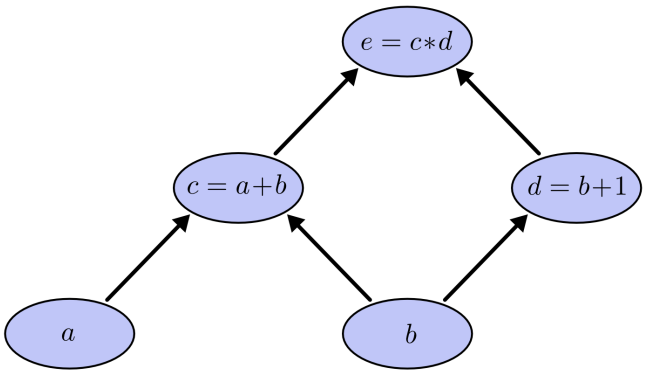

Figure 2.7: An example computational graph with inputs a, b and outputs $(a + b)(b + 1)$[7]

[7]http://colah.github.io/posts/2015-08-Backprop/

It is trivial to traverse the graph from all input nodes to calculate its outputs. If we model a neural network as a series of computations (eg. $y = sigmoid(x \cdot w)$), then this corresponds to the network's forward pass. Conversely, since each node knows about its outcoming edges and operator, it can infer the gradient w.r.t each edge. In other words, we can calculate gradients at each node automatically, by traversing the graph backwards. This is immensely helpful for backpropagation, which requires the gradient for each neuron to be calculated.

The fact that neural networks can be modeled as computational graphs has several implications. Most significantly, it means that backpropagation can be done automatically, assuming each op is differentiable. This greatly reduces the amount of work required when designing network architectures, as only the forward pass need to be defined. Moreover, graph optimization methods can be applied to their architectures, allowing them to be more compact and efficient.

2.3.2 Deep Learning Frameworks

The use of computational graphs for deep learning became prevalent in part due to the multiple frameworks that leverage them for automatic gradient calculations. Two of the earliest were Theano (2010) [2] and Caffe (2014) [11], followed by Google's TensorFlow (2016) [1], which remains the most popular framework to date. These tools define the same 2-phase workflow for running neural networks: the first phase where the computational graph is symbolically defined, and the second phase where numerical input is fed to the runtime to execute the graph.

By seperating graph definition and the actual runtime, these early frameworks can apply several optimizations during graph "compilation". This also helps improve performance, since definition is often done in the Python language, while the runtime can be written in lower-level, more performant languages (eg. C, C++,...). Finally, since the graph is finalized after compilation and is essentially data, they are portable and serializable, which helps deployment at scale.

On the other hand, compiled graphs are often criticized for their steep learning curve and being difficult to debug. Since the framework runtime completely takes over execution and even modifies the graph during optimization, it's often difficult to keep track of results and errors. This motivated the creation of dynamic graph frameworks, most notably PyTorch (2017) [19] and Chainer (2015) [31]. These frameworks simply constructs graphs *on the go*, alongside the actual calculations. This has the benefit of

being easier to grasp, and operations can be transparently observed instead of obscured during execution. However, they suffer from fewer optimizations and lower portablility.

Despite their differences, static and dynamic graph frameworks have shown a tendency to overlap in recent years. Namely, TensorFlow 2.0 introduced Eager Execution, which attempts to create a smoother, dynamic experimental workflow. In contrast, PyTorch 1.0 added Script Mode in order to support compiled graphs, a feature that used to be delegated to conversion to Caffe 2. In the end, we can expect these libraries to provide both an intuitive development experience and optimized for production.

On top of these frameworks, multiple tools and interfaces have been proposed in recent years to help simplifying their workflow. High-level libraries like Keras [5] allows intermediate users to interact more easily with TensorFlow, Theano and CNTK, while tools such as NVIDIA DIGITS create a streamlined workflow for multiple frameworks.

2.4 Training Neural Networks in Parallel

A major drawback for deep neural networks is the computational complexity of training them. LeNet (1998) [14], for example, has over 100,000 parameters, while ResNet-50 (2016) [10] has 25 million. Optimizing these parameters over many iterations is very computationally intensive, and training even a small network such as LeNet could take several days on standard CPUs.

From a parallel computing perspective, optimizing neural networks has a lot of potential optimizations. A common approach is running training examples in parallel. In each iteration, gradients are calculated for different data batches in each node, then reduced into the final update values for the model. The reduce procedure is critical in this approach, as it determines the communication overhead for each iteration. The most straightforward method would be using master nodes (often denoted parameter servers), where gradients are collected and reduced. However, this creates a bottleneck at the parameter servers themselves, and forces another level of communication if multiple parameter servers are used.

A more efficient method called "Ring Allreduce" was proposed by researchers at Baidu [24]. As the name suggests, this method places nodes sequentially into a "ring". When the reduce operation is called, the first node sends its gradient to node 2, where it's reduced with node 2's gradient. This result is then sent to node 3 and so on. When all nodes are reached, node 1 would contain the final gradient. We would then make

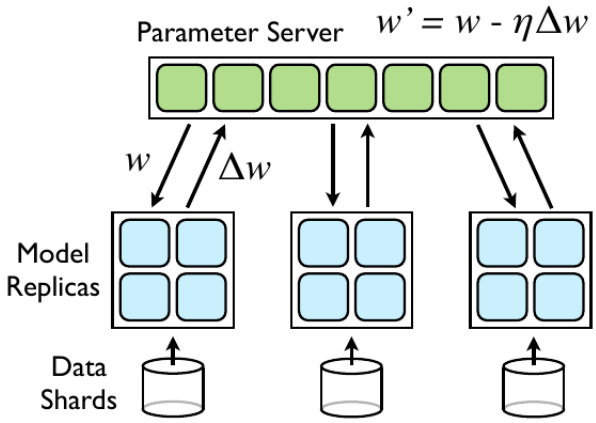

Figure 2.8: Centralized Allreduce with Parameter Server [6]

another pass around the ring to update all nodes with the new gradient.

Something to take note of is the effect of this paradigm on the training results itself. Since we are running each replica on different batches, the model is essentially training with a larger batch size. Multiple works, for example [3], have observed that huge batches can hinder a network's learning process, especially in early iterations. This puts somewhat of an upper limit to the scalability of this approach.

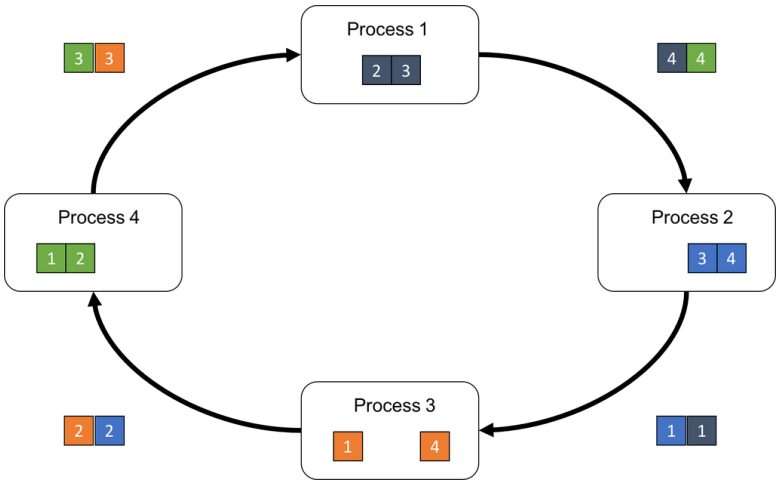

Figure 2.9: Ring Allreduce[8]

Another approach, as implemented by the authors of the DistBelief framework [6], is to distribute the network graph vertically on multiple nodes. This "striping" appproach avoids constant synchronization between nodes, but also only applicable to large, wide networks whose layers can be efficiently split.

Finally, a strategy called "pipeline backpropagation" was investigated by the authors in [20]. The idea is similar to striping, where the network graph is distributed across

[8]https://preferredresearch.jp/2018/07/10/technologies-behind-distributed-deep-learning-allreduce/

nodes. However, this method partitions the graph horizontally, where each node contains one or several full layers, forming a data pipeline. During training, each node works independently on their own data, disregarding the current state of the whole graph. Updates are performed with a delay, which become more noticeable at deeper layers of the network.

2.4.1 Using Graphics Processing Units

A breakthrough for training DNNs was the use of Graphic Processing Units (GPUs), as first presented by Raina et al [21] in 2009. Similar to graphics processing, neural networks are trained using operations on large matrices, which benefit from GPUs' multi-core design. Since then, the toolchain for GPU-accelerated neural network has matured significantly. Libraries such as CUDA, CUDNN [4], and high level interfaces have made training of neural networks on GPUs trivial.

All of the frameworks mentioned in 2.3.2 allows execution on GPUs, the majority of which rely on NVIDIA's CUDA and CUDNN libraries.

2.4.2 Using Computing Clusters

While GPUs have greatly improved DNN training speed, modern networks continue to be even more complex, along with larger training datasets. This calls for more scalable training mechanisms, particularly on multiple devices. One approach, as seen with Google's Tensor Processing Units (TPUs) [12], is building custom hardware that are designed to work as clusters. Indeed, TPUs are very efficient and performant. However, this approach is expensive and requires a lot of effort for deployment. A more compact, user-friendly approach is to make use of existing GPUs and computer architectures. This can be done using a software layer that handles communication between GPUs/CPUs on different machines.

Support for multi-node and multi-GPU training are available in most DL frameworks, most notably TensorFlow (which utilizes Google's gRPC protocol) and PyTorch (which support several communication backends, including the popular MPI [7] interface). Both implement parallelization using the data parallel approach (with Allreduce), as this is the most general solution that can apply to most neural networks.

In 2018, researchers at Uber released Horovod [27], a library providing a unified interface for parallel training with TensorFlow, Keras, PyTorch and MXNet. Horovod

utilizes MPI for communication, and implements Baidu's Ring Allreduce. Horovod's notable feature is that it requires minimal changes to existing single-node codebases.

Chapter 3

Proposed Framework

3.1 BK.Synapse - A Framework for Distributed Neural Network Training

In this section, we present a tool and framework for training neural networks in a distributed manner called BK.Synapse.

3.1.1 Overview & Motivation

BK.Synapse provides a complete set of tools for training neural networks, including model and dataset managements, training configuration, monitoring and exporting results, combined with an intuitive, no-configuration distributed environment.

The use-cases and general design for BK.Synapse are inspired by NVIDIA DIGITS[1] (Deep Learning GPU Training System). DIGITS' primary use case is training deep neural nets on multiple GPUs, along with several computer vision-related utilities (eg. data visualization). The tool provides an user interface for interacting with and managing datasets, models, etc,... However, there are several downsides to DIGITS' design that motivated the creation of BK.Synapse, namely:

- Despite supporting Caffe, TensorFlow and LuaTorch, DIGITS' workflow is heavily optimized for Caffe. This forces major rewrites and reconfiguration for non-Caffe frameworks. For example, TensorFlow models need to be fully written within a single file, while subclassing a parent class that makes testing quite difficult.

[1] https://developer.nvidia.com/digits

- DIGITS was designed for single-machine, local deployment. This means it can only utilize multiple GPUs on the same machine, limiting scalability.

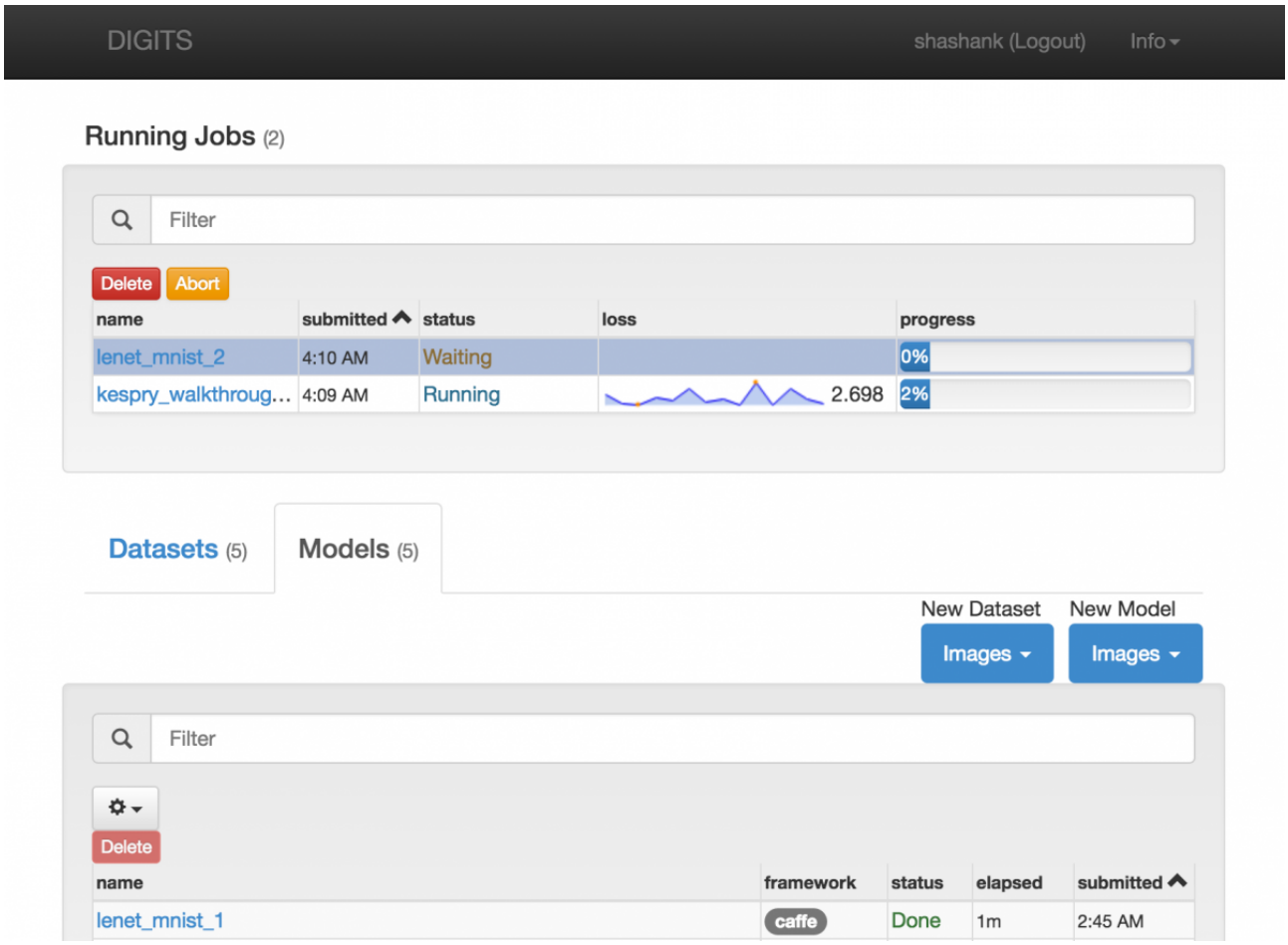

Figure 3.1: An example of DIGITS' user interface

Due to these limitations, we aim to design BK.Synapse as a distributed training framework with little to no development overhead. Users should be able to train networks on multiple nodes with ease, while making minimal modifications to their existing codebases. With this in mind, we also target the more user-friendly PyTorch as the primary deep learning backend, with planned support for Keras and TensorFlow in the future. Table 3.1 shows a detailed comparison between BK.Synapse and DIGITS.

Aside from DIGITS, we also take note of other similar tools in the domain, namely DeepCognition[2], Google Colaboratory[3], etc,...

BK.Synapse is open source and hosted at https://github.com/lanPN85/BK.Synapse.

We shall describe BK.Synapse's design and technical implementation in the following sections.

[2]https://deepcognition.ai/
[3]https://colab.research.google.com/

Feature	DIGITS	BK.Synapse
Single-node multi-GPU support	✓	✓
Job monitoring & management	✓	✓
Open source	✓	✓
Multiple node support	✗	✓
Arbitrary models and datasets	✗	✓
Data visualization	✓	✗
Model visualization	✓	Partial (via Tensorboard)
Backends	Caffe, TensorFlow, LuaTorch	PyTorch, Keras (planned), TensorFlow (planned)

Table 3.1: Feature comparison between DIGITS and BK.Synapse

3.1.2 Core Concepts

There are 4 core concepts that define the user's workflow in BK.Synapse (Figure **??**):

1. Datasets: A dataset is a collection of arbitrary files used for training (eg. images, text files,...). There is no pre-defined format, and users can simply compress their existing data and upload to the server.

2. Models: A model contains the code for the user's custom network. In order to facilitate dynamic loading during runtime, this code needs to adhere to a simple interface that will be described in later sections.

3. Data loaders: A data loader is the glue component that connects a dataset and a model. Conceptually, data loaders are the same as PyTorch's Dataset or Keras' Sequence. They define how data from datasets should be loaded and transformed in order to be fed into models.

4. Jobs: A job puts the above components together to create a complete training pipeline. It also specifies many hyperparameters for training (eg. learning rate, batch size,...) and the nodes to be used for the training process.

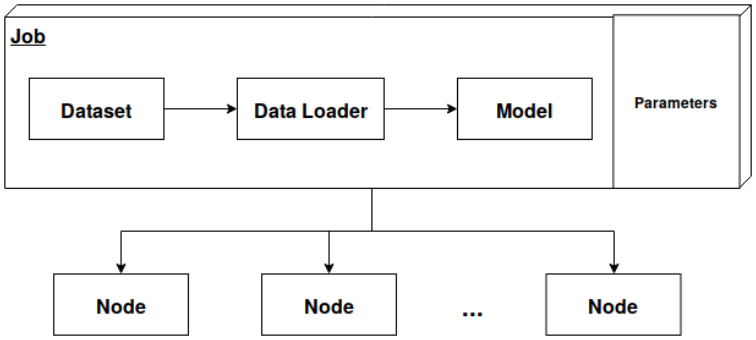

Figure 3.2: BK.Synapse' core concepts and workflow

3.1.3 Architecture

At a high level, BK.Synapse consists of 4 primary components (Figure 3.3): the API server, worker nodes, data store, and the web application.

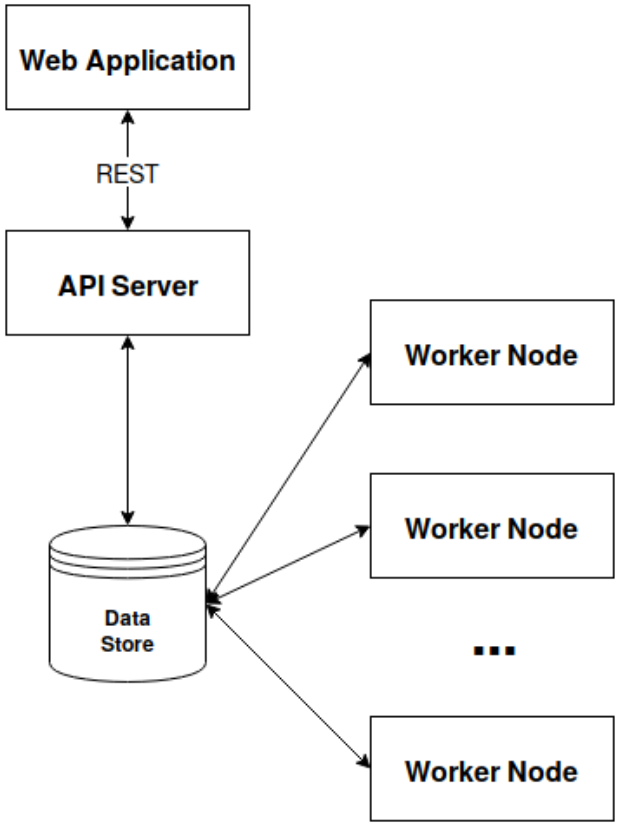

Figure 3.3: BK.Synapse's high-level architecture

The system is designed in a client-server pattern, with loosely coupled components that can be deployed separately. The deployment environment should have high availability (ie. nodes should not fail or shutdown regularly) and trusted.

The API server exposes an application programming interface via HTTP REST[4]. Clients can use standard HTTP requests from any supported language to access the system. The server is implemented with the Flask [9] framework.

Worker nodes handle the bulk of calculations during training. Each node runs a background process called the node daemon, collecting hardware status and notifying BK.Synapse of the node's availability.

The data store (or root folder) is used as the shared data space between the API server and worker nodes. It points to a folder that is accessible from all nodes within the system, and contains all user-uploaded data as well as additional metadata. Several mechanisms can be employed to create the data store, namely:

- Using filesystem mount: The data store is created on one node, then mounted throughout the network using Linux's NFS or SMBD protocol. This provides good guarantees in terms of writes, however there may be overhead during training when data files need to be continually read.
- Using object store mount: Cloud object stores such as Amazon S3 or Minio provide utilities that allows mounting onto a folder. This can be significantly less trivial to setup compared to the first option, however it can perform much faster depending on the object store implementation.

The primary reason to use a native file folder as data store, as opposed to an object store or database is due to accessiblity. As stated, BK.Synapse aims to be simple and user-friendly, and thus should expose a simple folder structure to users instead of a complex data management system. With our setup, users can reuse their data loading logic directly into their BK.Synapse code.

Finally, the web application is a client that uses the BK.Synapse API, providing users with access to the system via a graphical interface (Figure 3.4). It also includes documentations and various examples of using BK.Synapse. The application is built using the Vue[5] framework.

[4]Representational State Transfer
[5]https://vuejs.org/

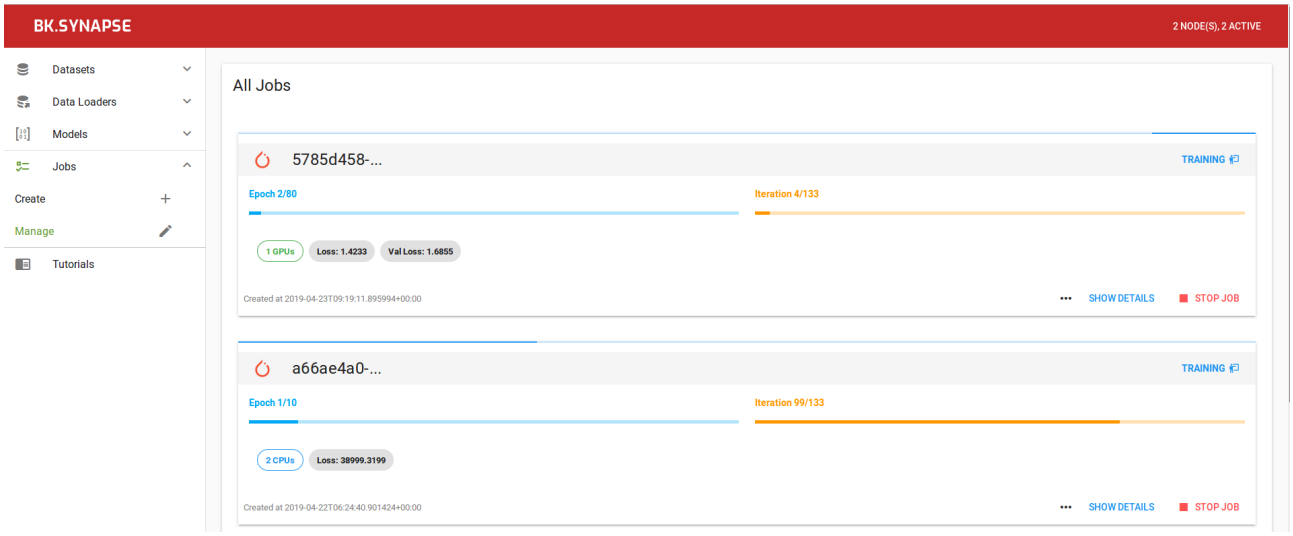

Figure 3.4: The BK.Synapse web application

3.1.4 Object Management Model

BK.Synapse manages objects directly through the filesystem within the shared datastore. The directory structure is shown in Figure 3.5.

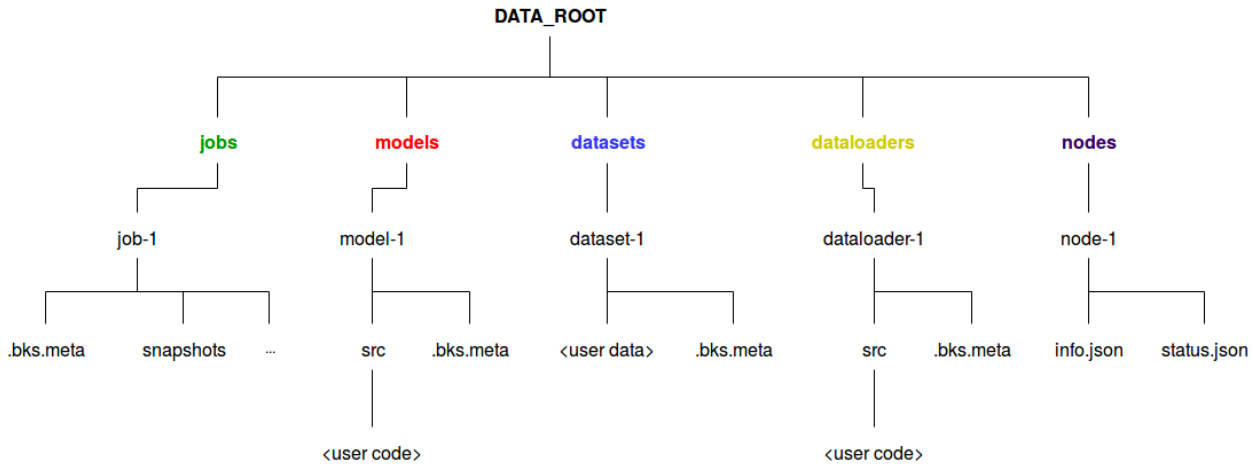

Figure 3.5: The BK.Synapse directory structure

Each object is assigned its own folder. Aside from nodes, whose metadata is provided by the node daemon in `info.json`, every object stores metadata in a `.bks.meta` file. This is a small JSON file that helps initialize these objects.

Models and data loaders store user-uploaded code in the `src/` folder.

The API provides several standard operations on each object type, including get queries, update, create, and delete. For uploading code and data, the user is required to put all source files and folders into a zip file, which is decompressed on the API server

into the datastore.

Even though a lot of metadata are stored as JSON documents, we chose not to use a full-fledged NoSQL database like MongoDB or a SQL database like PostgreSQL. Our arguments for this decision include:

- Consistency: since datasets, model code and loader code need to be placed in a datastore folder, it makes sense to store all data there as well, turning the datastore into a communication hub for the entire system. Using a database would introduce consistency constraints and require various checks and failure scenarios, while potentially slowing down overall performance.
- Simplicity: JSON files are easily read and accessed, which helps keep configuration to a minimum and eliminates complex database integration code. This also provides portability, as the entire system state can be easily packaged and replicated.
- Redundancy: With its current features, BK.Synapse would gain little from using databases. Most of our queries are simple get queries that might take even longer when put into databases. However, future improvements such as full-text search would compel us to use a database as a mirror to enable efficient queries.
- Finally, this decision is somewhat influenced by DIGITS, which uses a similar directory model to store jobs, albeit with a less portable file format (Pickle).

3.1.5 Dynamic Code Loading

The core BK.Synapse runtime is capable of embedding user-defined classes and functions across multiple files/packages into the training procedure. This requires the user to define entrypoints that the runtime can detect. These include:

- A file named `bks_model.py`, containing the class `UserModel` to define a model. `UserModel` must implement a `loss()` method that defines the loss function used in training. Optionally, if a `metrics()` method is defined, then it is used to calculate additional metrics (eg. accuracy). For PyTorch models, `UserModel` must subclass `Module`.
- A file named `bks_dataset.py`, containing the class `UserDataset` and (optionally) `UserValDataset`, which define the loading logic for a data loader. PyTorch-compatible loaders need to subclass PyTorch's `Dataset` class.

These interfaces are meant to wrap around existing code using inheritance, thus preserving their original behavior and facilitate testing, while still conforming to BK.Synapse's specifications. Simpler models and loaders can simply be written directly into the given class.

Listing 3.1: An example UserModel wrapped around a simple CNN

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
# ConvNet with 2 convolution blocks followed by dropout
# and 2 fully connected layers
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x

class UserModel(Net):
    def loss(self, output, target):
        return F.cross_entropy(output, target, reduction='mean')

    def metrics(self, output, target):
        preds = torch.argmax(output, dim=1)
        total = preds.shape[-1]
        correct = (preds == target).float()
        acc = torch.sum(correct) / total
        return {
            'accuracy': acc.item()
        }
```

The mechanics for embedding user code is relatively simple, and is partially based on DIGIT's method for loading TensorFlow models. We use Python's built-in `exec()` function, which can execute a string as Python code within the current environment. A major difference in our implementation is allowing relative imports from within the model directory. Since actual projects would have their code split into multiple files, this is a critical feature to ensure ease of use. We accomplish this by manipulating Python's path object (`sys.path`). The Python path is a list of filesystem paths that informs the interpreter where to look for modules. By inserting the source code root path to Python's path, we can emulate the user's environment and allow relative package loading. Once the neccessary modules are loaded, the injected path is removed to avoid pollution and/or conflict (Figure 3.6).

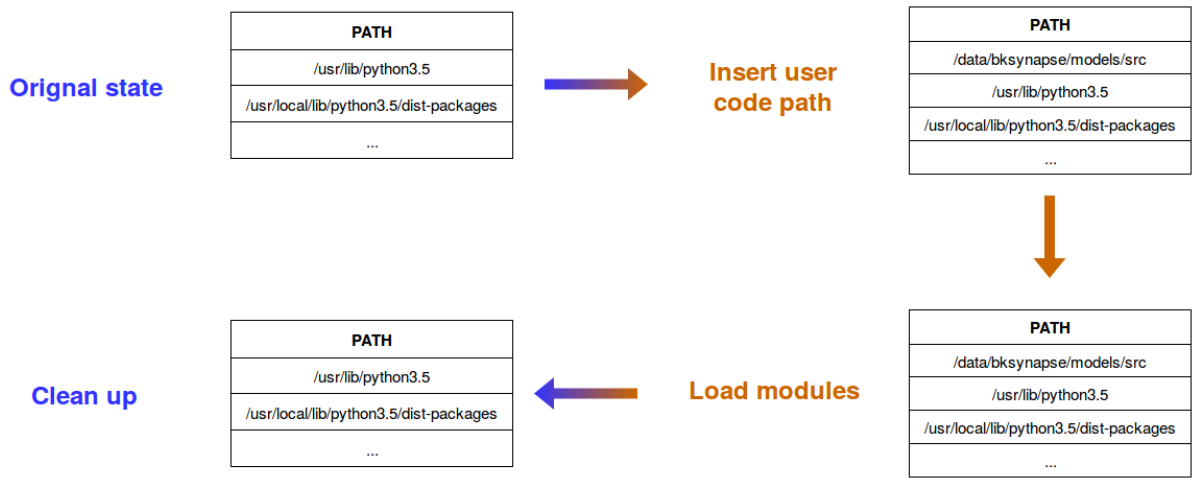

Figure 3.6: Dynamic code loading procedure

3.1.6 Job Monitoring

The user's custom model is loaded inside a training script that handles more than model training. Instead, it periodically logs progress information into a status update file (`status.json`). Clients can query the corresponding job's endpoint to receive this file's contents, and be notified of the training process. Additionally, status updates are also appended to a `history.toml` file, which serves as the job's history logs and used for future analytics. As job histories tend to be very long, we opt to store it as a TOML[6] array instead of a JSON document. On top of being human readable, TOML arrays can be easily parsed as a file stream without loading the whole document, which immensely improves query speed.

[6]https://github.com/toml-lang/toml

Listing 3.2: An example history.toml file

```
# ===Start===
[[entry]]
state = "SETUP"
isActive = true
epoch = 0
isStopped = false
totalIter = 0
timestamp = "2019-04-22T03:25:25.741491+00:00"
iter = 0
metrics = []
# ===End===

# ===Start===
[[entry]]
state = "TRAINING"
isActive = true
epoch = 0
isStopped = false
totalIter = 0
timestamp = "2019-04-22T03:25:36.699942+00:00"
message = "Starting␣training..."
iter = 0
metrics = []
# ===End===
```

The BK.Synapse web client polls the API backend every few seconds and update the monitoring UI. Even though this is not as real-time as some other solutions (eg. WebSockets), we argue that there's no penalty for slightly slower progress updates. In this case, we find that the polling model creates a more consistent and stable interface to our system.

The training script also generates a TensorBoard log directory for further analysis, and handles saving checkpoints at regular intervals.

3.1.7 Node Monitoring

Similar to jobs, nodes are monitored using node daemons. A node daemon is deployed at each node, and is in fact a simple Python script that runs an infinite loop. Within the loop, the daemon i) queries CPU and GPU information, ii) updates the node status and iii) waits several seconds to the next loop. Constant information like GPU names, CPU

frequency, etc... are collected on startup. Only status information, eg. RAM usage, GPU memory usage,... are constantly queried. This information is also polled and displayed by the BK.Synapse web app.

Each status update also contains a timestamp, used by the API server as a health check mechanism. If a node's timestamp is "stale" (ie. too far away from the current timestamp), that node is declared inactive.

3.1.8 Parallelization Techniques

BK.Synapse's key feature is being able to scale easily across multiple machines. The infrastructure layer supports this by keeping track of node IPs using node daemons, making it easy to add new nodes or detect failures. The actual parallel execution is implemented with Horovod [27] and MPI [7].

Horovod provides a set of utilities to leverage MPI and associated acceleration methods (eg. NCCL) to perform multi-node training:

- An `hvd.rank()` function that provides the process MPI rank, as well as `hvd.local_rank()` to query local MPI rank.
- A `DistributedSampler` for PyTorch that partitions datasets into non-colliding sets for each process rank.
- A ring allreduce implementation (`hvd.allreduce()`) providing efficient value aggregation.
- A `DistributedOptimizer` wrapper for PyTorch that allows networks to be updated over multiple nodes.

The training script is invoked using `mpirun`, from a subprocess spawned by the API server when the start endpoint is called. The server collects the selected worker nodes' IPs and generate a command as follow:

```
mpirun --allow-run-as-root -bind-to none -map-by slot\\
    -mca plm_rsh_args "-p␣17992" -mca pml ob1 -mca btl ^openib\\
    -mca btl_tcp_if_include 192.168.1.0/24  -x LD_LIBRARY_PATH\\
    -x PATH -x BKSYN_DATA_ROOT -v -np 2\\
    -H 192.168.1.2:1,192.168.1.8:1\\
    python /usr/bin/bksynapse/pytorch/train.py\\
        --job-id d20db7d2-2f22-4f86-bbc7-8e93c5239132
```

A lot of configuration is handled by BK.Synapse to ensure proper MPI communication between nodes, in regards to the `-mca` options above.

We make use of the rank 0 node as the logger, periodically writing the job's status to the shared folder. Interrupts are handled using a special lock file (`job.lock`) that gets checked at regular intervals.

For nodes with multiple GPUs, the default behavior is to use all those that are available, each getting its own MPI process. We use local rank to pin each process to its respective GPU.

3.1.9 Deployment Model

All components are packaged as Docker containers for ease of deployment. The only system dependency is the shared data store, which is subsequently mounted to the `/data` folder within each container.

MPI requires a single port to be used for SSH connection across all machines. To avoid collision, we use the port 17992, and bind each container to the host machine's network stack.

For optimal performance, we have several deployment recommendations:

- High-speed connections are preferred when setting up the node network(eg. InfiniBand). Note that this does not have to include the web server, since this is separate from the training runtime.
- Node GPUs should have the same or similar capabilities. Otherwise, GPU speed or memory would be capped at the job's lowest-tier GPU, leading to wasted resources.

Our deployment workflow aims for simplicity and minimal configuration, allowing new nodes to be easily deployed. Docker helps isolate a lot of complex environment requirements, including SSH keys which are shared among nodes. Only a few system dependencies need to be setup, including:

- GPU drivers: this component is shared in the NVIDIA-Docker runtime model, and needs to be installed for GPU nodes to function.
- Shared drive mount: the datastore mount method is deployment-specific, and needs to be manually configured.

- Networking configurations: several configurations related to network addressses have to be configured, as detecting correct configurations would be too unreliable. These values are mostly related to various IP addresses, and can be quite easily queried.

3.1.10 Security Concerns

In its current state, BK.Synapse should only be deployed in a trusted, private environment. This is due to a number of security concerns, including:

- No access control or authorization has been implemented. This is however an important feature that will eventually be added once the API is more stable.
- The use of `exec()` can create many attack vectors. Since `exec()` would execute any Python command, a malicious agent can freely inspect or modify the system. Even though Docker provides some level of isolation, attackers can still easily cripple nodes or retrieve secret data. These vulnerabilities can be patched by setting up less-privileged users, or blocking certain modules from loading.

Since this is somewhat out-of-scope, we will not go into detail on this topic.

3.2 Case Study: RetinaNet for Text Region Detection

To fully evaluate BK.Synapse's effectiveness, we investigate its use in training RetinaNet, an existing CNN architecture for object detection.

RetinaNet [16] is a network architecture proposed by Lin et al in 2017. RetinaNet is a one-stage detector in similar spirit to YOLO, however it aims to match two-stage detectors in terms of accuracy while maintaining inference speed.

3.2.1 Feature Pyramid Network

RetinaNet improves primarily on Feature Pyramind Networks (FPN) [15]. FPN works on the intuition that it is better to take advantage of several feature maps (layer outputs) of a network for prediction, instead of just the final output. The reasoning for this is quite simple: smaller objects benefit from large, coarse feature maps where they are more visible, while larger/more complex objects are more easily detected in small, fine

feature maps where they are more simplified. The network design is described in Figure 3.7.

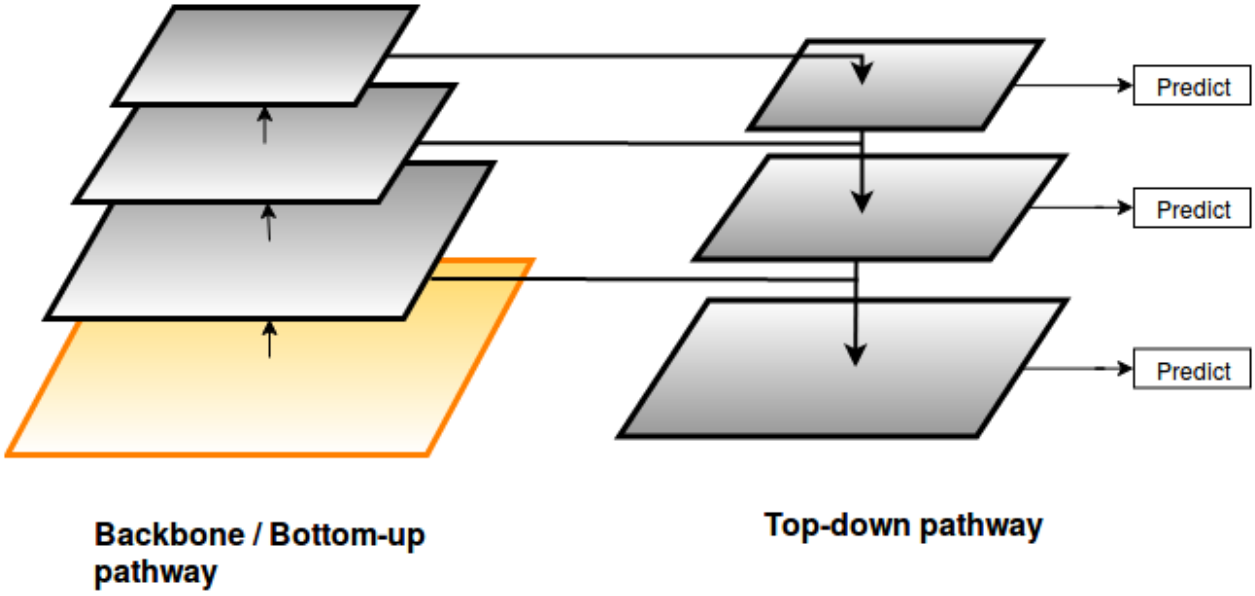

Figure 3.7: Feature Pyramid Network

FPN utilizes a secondary CNN architecture called the backbone. The backbone network can be any convolutional neural network, and is typically a commonly used architecture (eg. VGG, ResNet,...). The backbone's convolution blocks make up the bottom-up pathway, where feature maps of different layers are created. The authors define network "stages", in which convolution blocks have the same output size. In the case of ResNet, the authors used the activated output of each stage's residual block as feature maps.

A top-down pathway is used in conjunction with a conventional regressor/classifier pair. The final feature map is upsampled (using nearest neighbor upsampling) and added element-wise to the corresponding lower-level feature map from the bottom-up pathway. The bottom-up map undergoes 1x1 convolution to generate uniform channel dimensions, while the merged map also goes through a 3x3 convolution layer to reduce aliasing. This addition operation forms lateral connections, similar to UNet-type architectures. The resulting feature map combines high-level features from later layers, as well as coarse features from the previous layer. It is then fed directly to the shared regressor/classifier heads and produces predictions.

FPN places anchors at each prediction feature map. Each feature layer has gradually larger anchor sizes, with multiple aspect ratios. Labels are assigned to anchors based on its IoU value compared to ground truth boxes. Specifically, if a ground truth box

has IoU > 0.7 to an anchor, the anchor is labeled positive for that box, while an IoU < 0.3 marks a negative. Anchors with no matching boxes are marked as the special "background" class.

The classifier head attempts to classify each anchor to its correct class (either a foreground class or background). This is implemented as a simple 5-layer convolutional network, each with 3x3 kernels and 1 padding (which maintains the output shape). The final convolution layer outputs a tensor whose depth dimension equals $num\_anchors \times num\_classes$ and activated by the *sigmoid* function. This is used as the classification probability for each anchor (Figure 3.8).

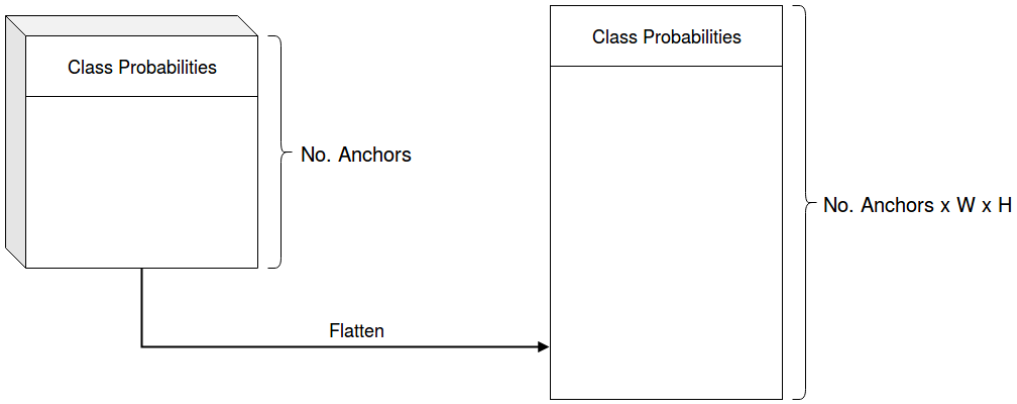

Figure 3.8: Transforming classifier head output

The regressor head serves to offset the anchor position to the ground truth box. It's implemented similarly to the classifier, however the final layer outputs a tensor with depth $num\_anchors \times 4$ and no activation. The 4 values represent the x, y offset values for the top left and bottom right corner of each anchor (Figure 3.9).

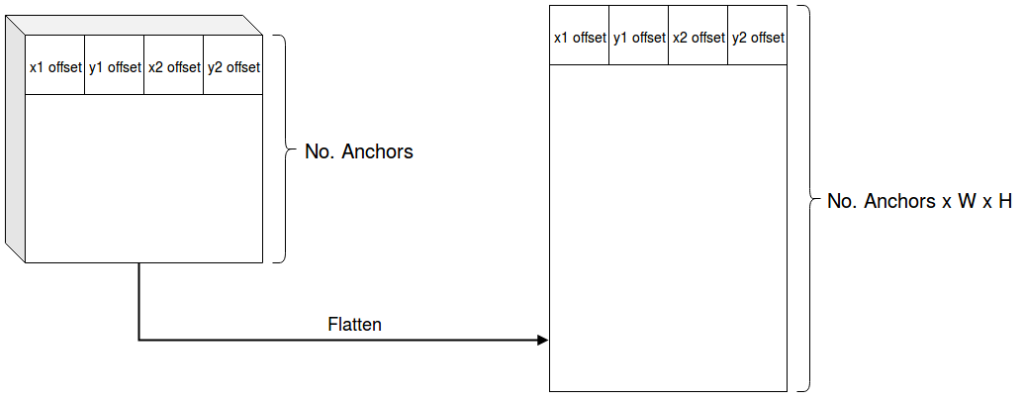

Figure 3.9: Transforming regressor head output

Output boxes are generated by taking positive foreground anchors and adding their box offsets. Overlapping boxes are merged using non-maximal supression.

Both the classifier and regressor heads are shared among all layer feature maps. The authors in [15] found that using different heads for each layer only showed marginal improvements. This implies that even at different levels of embedding, the learned features are similar enough for a single classifier/regressor to learn from.

In [15], the authors also applied FPN to Faster R-CNN, a two-stage detector, where it functions as the region proposal network. Each target region is mapped to a single feature level, determined by its size. Larger ROIs are put into finer feature maps.

3.2.2 Focal Loss

RetinaNet employs the same bottom-up to top-down architecture as FPN. The primary improvement in RetinaNet is a novel loss function called Focal Loss. Focal Loss builds upon Balanced Cross Entropy, a modified version of the Cross Entropy function used to handle large class imbalances:

$$CE(p_t) = -\alpha_t log(p_t) \tag{3.1}$$

where α_t is a weighting factor determining how much to "focus" on negative examples as opposed to positives. In essence, α_t controls which class is prioritized during the training process. Focal Loss goes one step further and seeks to control how much to prioritize "hard" examples:

$$FL(p_t) = -\alpha_t(1 - p_t)^\gamma log(p_t) \tag{3.2}$$

where γ is the focusing parameter. We can see that by adding the modulating factor $-(1-p_t)^\gamma$, an example's loss diminishes as the model's prediction approaches the ground truth. In other words, *easier* examples are down-weighted, therefore allowing models to focus on hard examples. As γ increases, the model focuses more on harder examples. This transition is smooth, as seen in Figure 3.10.

Aside from Focal Loss, RetinaNet makes several minor adjustments to FPN. Different anchor scales are used at each layer for denser anchor coverage. IoU thresholds when assigning anchor labels are also relaxed, with a positive threshold of 0.5 and negative threshold of 0.4.

RetinaNet was shown to achieve better results on the COCO dataset [17] than Faster-RCNN, while also being faster.

When applied to the text detection problem, we take special note of the anchors. Unlike regular objects, which often has a ratio from 1:2 to 2:1, text blocks are usually

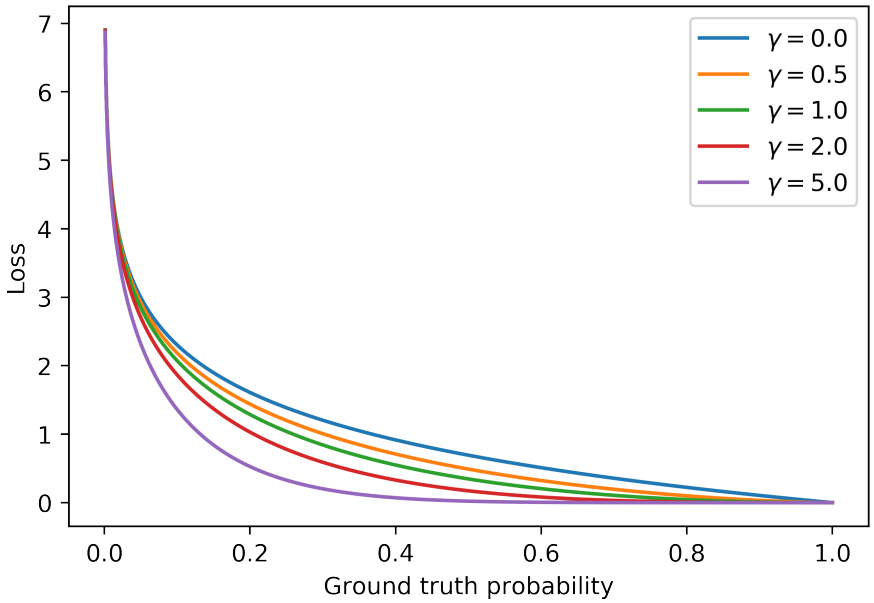

Figure 3.10: Focal Loss values over ground truth probability for different values of γ

wide. Therefore, the anchors used on COCO in [16] would not perform well on this problem. Instead, we add wider ratios of 1:8, 1:6 and 1:4, which better fit our data.

Chapter 4

Results & Evaluation

4.1 Experiment Setup

To benchmark BK.Synpase's performance, we trained the presented RetinaNet architecture using the framework. The RetinaNet implementation is written in PyTorch, based on an open source implementation and slightly modified to run on both CPU and GPU for testing purposes. The modified code is available at:
https://github.com/lanPN85/pytorch-retinanet-1

Due to resource constraints, we were only able to test the system on a small 2-machine setup. Both machine runs on Ubuntu Linux 18.04, which we denote as N1 and N2. They are connected via an Ethernet connection over a 100Mbps network switch. The shared data folder is on an HDD drive physically connected to N1, and mounted on N2 using Linux NFS mount. Each machine runs a BK.Synapse node daemon, while N2 also hosts the API server and the web application (Figure 4.1).

The training and test datasets are from ICDAR 2019 RobustReading Challenge on

Component	**N1**	**N2**
CPU	Intel Core i7-8700K, 3.70GHz	Intel Core i7-6700K, 4.00GHz
CPU Cores	6	4
RAM	64GB	32GB
GPU	NVIDIA GeForce GTX 1080Ti	NVIDIA GeForce GTX 1080
GPU Memory	12GB	8GB

Table 4.1: Hardware specifications for nodes N1 and N2

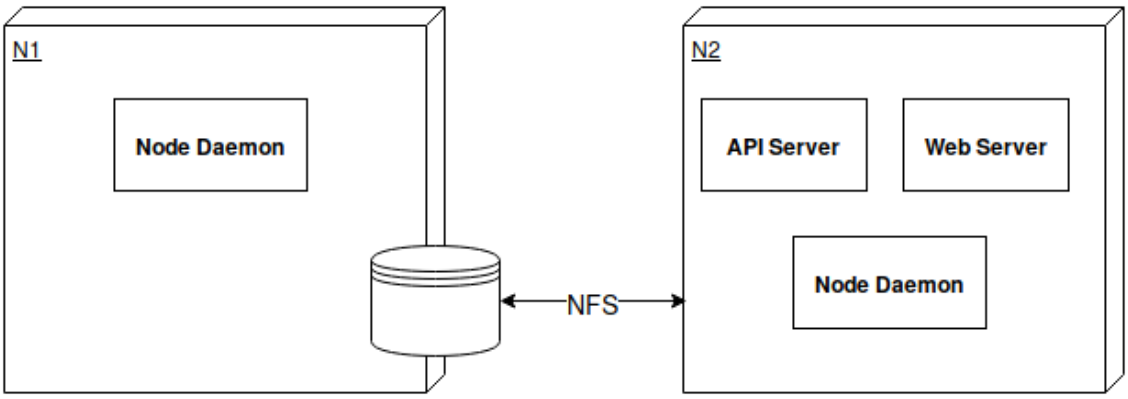

Figure 4.1: Deployment diagram for our experiments

Scanned Receipts OCR and Information Extraction, Task 1[1]. The original dataset consists of 626 images of scanned receipts, with annotations for invidual text block positions and their content. We split this data into 532 images for the train set and 94 images for the validation set. While the original challenge is to both locate each block and predict their content, we focus solely on the task of locating blocks using RetinaNet.

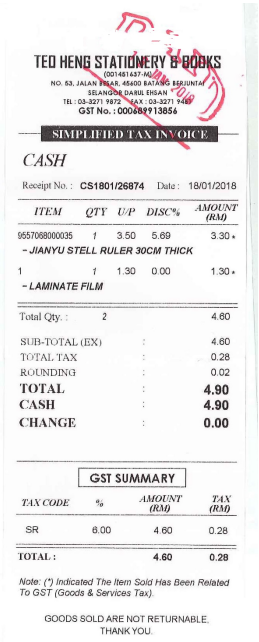

Figure 4.2: An example scanned receipt from the ICDAR Challenge dataset

[1]http://rrc.cvc.uab.es/?ch=13

Parameter	Value
Network backbone	ResNet-50
Learning rate	0.001
Batch size	2
Gradient norm	0.1
No. epochs	10
Snapshot frequency	2

Table 4.2: Benchmark training configuration

4.2 Training Benchmarks

Our first set of experiments seek to benchmark BK.Synapse's performance when training a large, production-level architecture. We trained the network with 4 different resource configurations: 1 CPU, 2 CPUs, 1 GPU, and 2 GPUs. The training configurations for each run are identical as shown in Table 4.2.

The listed batch size (2) is the per-node batch size. This value is quite smaller than normal, due to the size of the network and the input images. We find that a batch size of 2 avoids out-of-memory errors for our GPUs.

This benchmark is primarily concerned with parallelization metrics, including speedup and efficiency when training with multiple CPUs/GPUs, as shown in Table 4.3. We measure the average time in seconds for each epoch and step for comparison. Speedup and efficiency are measured using per epoch time.

As the 2CPUs/2GPUs setups work with larger batches, we see an increase in the average step time compared to single-node setups. For CPU runs, step time increases from 7.92s to 10.41s, whereas GPU runs increase from 3.04s to 4.48s. However, since epochs are now shorter, the overall epoch time is sped up. ON average, a CPU epoch decreases by approximately 700s (11 minutes), and a GPU epoch decreases by 200s

Resources	Avg. epoch time	Avg. step time	Speedup	Efficiency
1 CPU	2130.33s	7.92s	-	-
2 CPUs	1421.10s	10.41s	1.50x	75.00%
1 GPU	832.79s	3.04s	-	-
2 GPUs	606.14s	4.48s	1.37x	68.70%

Table 4.3: Benchmark time, speedup ratio and efficiency index for training RetinaNet in parallel

(3 minutes). a On 2 GPUs, BK.Synapse achieves a somewhat average efficiency index of 68.70%. On 2 CPUs, efficiency is slightly higher at 75.00%. This leaves quite some room for improvements, especially in terms of framework-specific tasks like logging and monitoring. However, the training speedup is still significant, and can be very useful in practice. Figure 4.3 shows the average epoch time between these configurations and the "ideal" runtime (ie. efficiency equals 100%).

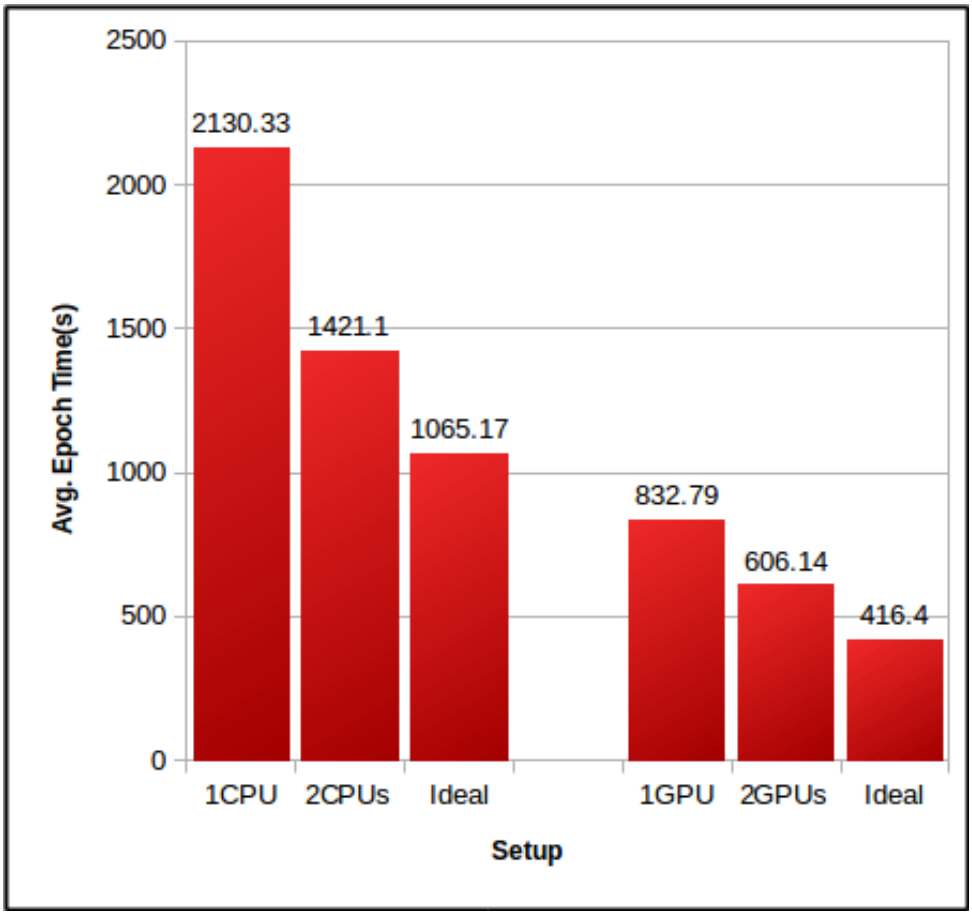

Figure 4.3: Average epoch time for different parallel configurations

4.3 Model Performance

In our final experiment, we train the same network to convergence using BK.Synapse, and evaluate the final model's performance on the ICDAR Challenge task. We maintain the same parameters as in Table 4.2, but set the number of epochs to 50. The network is trained on both N1 and N2 with GPU acceleration.

The model's loss values (regression and classification loss) at each epoch for the training and validation sets are shown in Figures .

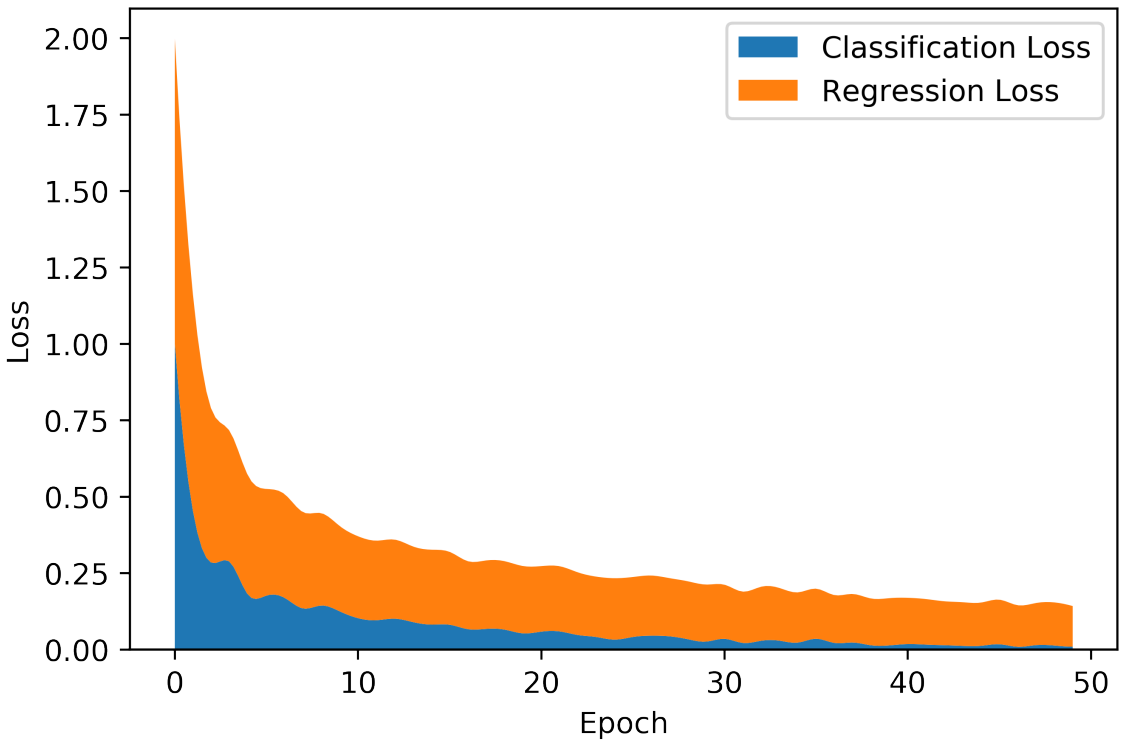

Figure 4.4: Regression loss and classification loss over epochs on the training set

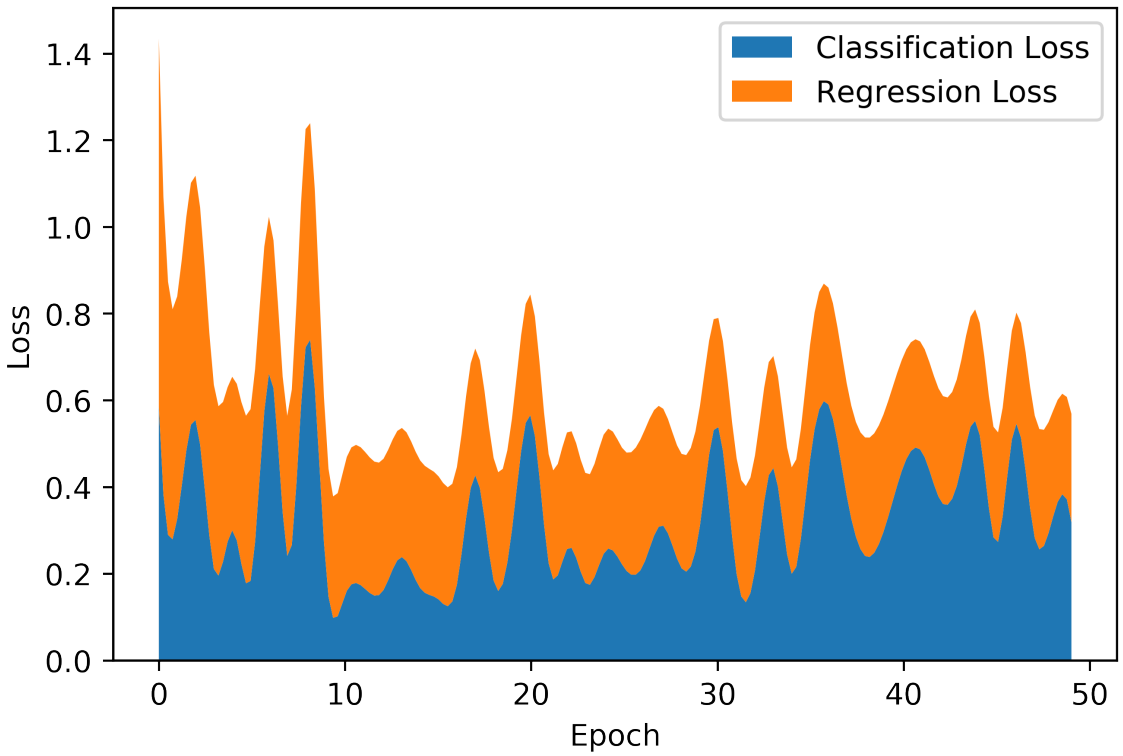

Figure 4.5: Regression loss and classification loss over epochs on the validation set

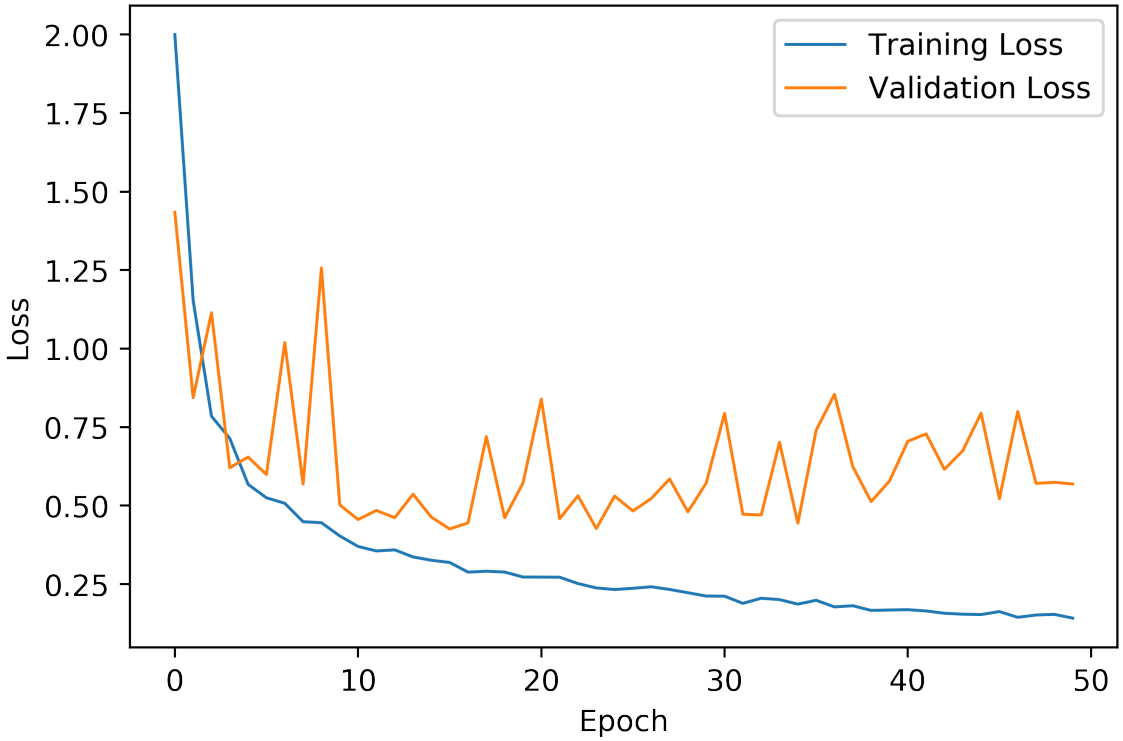

Figure 4.6: Loss value comparison between the training and validation set

Chapter 5

Conclusions

This thesis has presented BK.Synapse, a framework for distributed neural network training. Each component has been described in detail, with heavy emphasis on user workflow and integration with existing code. We also showcased the tool with a case study on RetinaNet for object detection, and received positive, if somewhat early-stage, results.

At the same time, there is much room for improving BK.Synapse, which motivates our future development efforts. Our future works include:

- Improving security and multi-user workflow, with the goal of having a viable public deployment.
- Adding support for the planned deep learning frameworks (Keras and TensorFlow).
- Adding more features in terms of monitoring and customizing the training process.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4. Austin, TX, 2010.

[3] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. Pipelined back-propagation for context-dependent deep neural networks. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[5] François Chollet et al. Keras, 2015.

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[7] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[9] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 1st edition, 2014.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[12] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[13] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[14] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[15] Tsung-Yi Lin, Piotr Dollar, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017.

[16] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[17] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[18] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[19] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*, 2017.

[20] Alain Petrowski, Gerard Dreyfus, and Claude Girault. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks*, 4(6):970–981, 1993.

[21] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM, 2009.

[22] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[23] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[24] Baidu Research. Baidu ring allreduce, 2017.

[25] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[26] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[27] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[28] David Silver and Demis Hassabis. Alphago: Mastering the ancient game of go with machine learning. *Research Blog*, 9, 2016.

[29] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[30] Tijmen Tieleman and Geoffery Hinton. Rmsprop gradient optimization. *URL http://www. cs. toronto. edu/tijmen/csc321/slides/lecture_ slides_ lec6. pdf*, 2014.

[31] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[32] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.