Contents

Introduction 5

An Introduction to Neural Networks 13

Modern Neural Network Development 47

Taffy, a Compositional Representation 59

TaffyStudio 79

Conclusion 89
Introduction

Abstract

Neural Networks remain inaccessible or difficult to a large number of users, in spite of tremendous efforts and offerings. While these offerings have increased in number and quality, Neural Network technology remains available to a just small number of users, who still find the process of creating Neural Networks onerous and error-prone. This work presents Taffy\textsuperscript{1} and TaffyStudio\textsuperscript{2}, a framework and visual interface that takes some of the best ideas from existing systems and combines and extends them to make the definition of Neural Networks radically easier, simpler, more interpretable and interactive.

Programming Constructs

Over the past three decades, systems have been developed to aid in the definition and execution of Neural Network models. The past decade has seen a surge in the use and popularity of Neural Networks, inspiring tremendous work in systems

\textsuperscript{1}https://github.com/julianoks/Taffy
\textsuperscript{2}https://github.com/julianoks/TaffyStudio
such as TensorFlow, Torch, Caffe, and others. Almost all of these systems abstract Networks as computational graphs composed of tensor operations.

Some systems interface directly to the graph data structure, like Caffe, while others implicitly construct graph data structures underneath a programming interface. Like the latter, the Taffy system exposes a programming interface as a dataflow graph of arbitrary operations, which is used to construct a graph of tensor operations. Separating the dataflow and tensor graphs allows for a higher-level interface.

**Modules**

Most popular systems now provide layers, which are higher-level operations composed of many lower level operations. Layers allow for the definition of models at a higher level of abstraction, simplifying the development process by making the definition simpler and more interpretable.

Further, systems like TensorFlow now provide even higher level constructs called "Estimators", which can take the form of entire models or loss functions. There exists a spectrum of abstractions, ranging from primitive operations to layers to blocks to entire models.

An important factor in the quality of a framework is the availability of layers and other pre-made constructs. Taffy is built upon the concept
of a module, a computational graph whose nodes may use other modules. In Taffy, every graph is a module that can be reused and composed with other graphs.

Modules span many levels of abstraction using a homogenous concept. Users can build and import modules that use operations ranging from simple layers, to larger blocks, to entire models.

Not only do modules make defining Networks easier, but they vastly simplify the graph layout by encouraging each module to be a relatively small collection of related operations.

**Visual Programming Environment**

TaffyStudio is a visual programming interface to Taffy, which differs from most other systems that expose a programming interface to a language like Python or C++. Like many other visual programming interfaces (Scratch, Blender Nodes, Houdini Nodes), TaffyStudio is meant to have a short learning curve, allowing users to build Neural Networks with minimal prior explanation. Despite this ease, Taffy is powerful and rational, with consistent and intuitive semantics that allow high-level operations to be sensically built and inspected.

Exposing a visual programming interface has several advantages over a text-based interface, like Python or C++ code. For these text-based interfaces, programmers must parse through the
code and mentally or diagrammatically form a picture of the graph structure. TaffyStudio’s visual interface makes the inherent graph structure immediately apparent. For example, figure 1 contrasts an implementation in TaffyStudio and in TensorFlow code.

Current state-of-the-art visualization libraries can only render these diagrams after compilation. Perhaps the greatest drawback of these visualization systems is that they are decoupled from the construction of the Network. TaffyStudio integrates the visualization and construction of a Network, providing live and continuous feedback with immediate results.

Annotation

Neural Networks apply tensor-valued operations over tensors of different types and dimensionality. Because these operations have various constraints and semantics, developers often annotate their source code and diagrams with the shape and type of tensor values. However manual annotation is error-prone and tedious, and so a small number of state-of-the-art visualization libraries, most prominently TensorBoard, perform this annotation for the user. Unfortunately, these visualizations are produced after compilation, and the annotations correspond to the underlying tensor graph rather than the user-provided code.

Because Taffy is already in an ideal graphical
format, TaffyStudio can directly annotate the user-provided representation. Unlike any other currently available system, TaffyStudio continuously annotates the graph as it’s being built, allowing users to view the correct annotations before the Network has been compiled and without interfacing to another visualization system.

**Debugging**

Debugging is one of the greatest challenges of current Neural Network systems. Most systems have a programming interface that throws a stack trace upon error. Not only are stack traces inherently unclear, but finding the root cause of an error is further obscured because other values, which may be causing the error, are not reported. Hence one must use print statements or a tool like the python debugger. However, traditional debugging may be useless because most libraries, like TensorFlow, use an internal structure that’s executed in a separate runtime, hence many values are unavailable to the exposed runtime. To alleviate these problems some of the more mature libraries have spent considerable effort developing ad-hoc debugging interfaces, such as the TensorFlow debugger.

Because Taffy is not responsible for materializing values and is built up from primitives, debugging occurs in a single runtime and errors can be attributed to specific primitive operations.
nostic error messages are provided at the node where the error occurred. To diagnose and fix the error, a user can then inspect the annotated values being supplied to that node and other values in the graph.

To account for the hierarchical structure imposed by modules, TaffyStudio supports nested debugging. If an error occurs at a node that uses a module, its error message has a link to step into the module where the error occurred. Within this module, the values present at runtime are annotated, allowing users to inspect the state at the time of the error. That module’s error may in turn link to another module, allowing nested debugging to drill arbitrarily deep.

TaffyStudio continuously debugs a module as it’s being developed, encouraging users to correct errors as they’re being made. This fast feedback coupled with superior debugging and annotations provides an easier, more robust and informed development process.

Compilation

Most systems are used to both create and execute a Neural Network. In contrast, Taffy defines a Neural Network and compiles it to another backend, like TensorFlow. Taffy compiles to a very simple intermediate representation with relatively few primitive operations, making it easy to write custom compilers for arbitrary backends.
Simple rewrite rules constitute the majority of such a compiler, typically implemented in a dictionary of one-liners. With recent work in interoperable formats, specifically ONNX, accurately compiling to a large number of popular backends is becoming easier.

Contents

Chapter 1 provides an extensive overview of Neural Networks. First, the anatomy of a Neural Network will be discussed, followed by a thematic discussion of larger architectures, and finally the learning algorithms used to optimize the Neural Network’s performance. An emphasis will be placed on the mathematical and computational foundations, including equations and implementation details.

Chapter 2 will survey popular Neural Network systems. The survey will identify and compare the systems’ internal representations, as well as interfaces to Neural Network construction and learning procedures. It is shown that almost all systems use a similar graphical representation, although differ in terms of interfaces, semantics, and the definition of learning algorithm.

Chapter 3 introduces Taffy, a compositional representation for Neural Networks. Chapter 4 outlines TaffyStudio, a browser-based interactive visualization system for authoring Neural Networks in Taffy. Chapter 5 concludes the writeup,
offering lessons learned, future work and directions.
An Introduction to Neural Networks

A neural network is a universal function approximator composed of simpler functions. Section 1 of this chapter focuses on the general structure of a neural network, in terms of the functions it’s comprised of and their arrangement. Section 2 discusses some of the various learning algorithms used to "train" a neural network - that is the algorithms responsible for finding parameters that improve the network’s performance given a set of examples. Lastly, section 3 will outline Automatic Differentiation, the essential algorithm that enables many learning algorithms by efficiently calculating the gradient of the objective w.r.t. the parameters.

Structure of Neural Networks

In it’s simplest form, a neural network is an organization of simple processing units, sometimes called nodes or neurons. Each unit is a parameterized function \( f : \mathbb{R}^d \rightarrow \mathbb{R} \) whose inputs are the outputs of other units, and whose output is a
single scalar value. This function is calculated as:

\[ f(x) = \phi(w \cdot x + b) \]  

where \( x \in \mathbb{R}^d \) is a vector of input units, \( w \in \mathbb{R}^d \) is a vector of "weight" parameters, \( b \in \mathbb{R} \) is a "bias" parameter, and \( \phi : \mathbb{R} \to \mathbb{R} \) is a nonlinear **activation function**. Neural networks are organized into sets of units, called **layers**. Neural networks often contain **fully connected** layers, where every unit in the fully connected layer takes every unit in another layer as input, as shown in figure 2. A fully connected layer with \( n \) units whose input is a layer with \( m \) units could be considered a vector-valued function, \( f : \mathbb{R}^m \to \mathbb{R}^n \)

\[ f(x) = \phi(W^T x + b) \]  

Where \( x \in \mathbb{R}^m \) is a vector of inputs, \( W \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^n \) are the weight and bias parameters, and \( \phi : \mathbb{R}^n \to \mathbb{R}^n \) is an element-wise nonlinear activation function (described next). Generally, a neural network may contain other functions in addition to neurons, although neurons are typically the primary form of computation. In the next subsection, the workings of a neuron will be further explored by motivating and giving examples of activation functions.

**Activation Function**

In this subsection, the workings of a neuron will be further explored by motivating and giving examples of activation functions. First, it is
proven that a nonlinear activation function must be used to enable a prototypical neural network to represent nonlinear functions. A prototypical neural network is a neural network that can be expressed as a series of fully connected layers, \((f_n \circ \ldots \circ f_2 \circ f_1)(x)\).

**Lemma 0.0.1** (Composition of linear functions is a linear function). For two functions:

\[
\begin{align*}
  f_1(x) &= w_1x + b_1 \\
  f_2(x) &= w_2x + b_2
\end{align*}
\]

where \(w, x, b\) may be scalar, vector, or tensor valued.

\[
\begin{align*}
  f_3(x) &= (f_2 \circ f_1)(x) \\
  &= f_2(f_1(x)) \\
  &= w_2(w_1x + b_1) + b_2 \\
  &= (w_2w_1)x + (w_2b_1 + b_2) \\
  &= w_3x + b_3
\end{align*}
\]

where \(w_3 = (w_2w_1)\) and \(b_3 = (w_2b_1 + b_2)\) are constants.

Thus, \(f_3\), the composition of two linear functions, is a linear function.

By induction, the composition of linear functions is a linear function. ■

**Theorem 0.0.2** (A prototypical neural network using only linear activations, or no activation, can only represent linear functions.).

Assume the case where \(\phi\) is a linear function \(\phi(x) = W_\phi x + b_\phi\). One is free to assume that the activation
function is element-wise, i.e. $W_\phi$ is a diagonal matrix, or that there is no activation, i.e. $W_\phi$ is the identity matrix and $b_\phi = \bar{0}$, as these are both linear functions.

Then a layer function is a composition of two linear functions, $\phi$ and $f(x) = W^T x + b$. By Lemma 0.0.1 a layer function is a linear function.

The composition of layer functions, $(f_n \circ \ldots \circ f_2 \circ f_1)(x)$ is also a linear function by Lemma 0.0.1. Hence a prototypical neural network using only linear functions must be a linear function. ■

**Corollary 0.0.2.1** (A nonlinear prototypical neural network must have a nonlinear activation function). Assume the negation: There exists a nonlinear prototypical neural network without a nonlinear activation function.

If a network does not have any nonlinear activations, it must only have linear activations, and must be a linear function as per theorem 0.0.2. Hence the negation is false, and by contradiction, the corollary is proven.

Now two of the most popular activation functions will be discussed: the ReLU activation function (Rectified Linear Unit), and the sigmoid function, denoted by $\sigma$. Their formulas are given by:

$$\text{ReLU}(x) = \max(0, x)$$

$$\sigma(x) = \frac{e^x}{e^x + 1}$$

The ReLU activation function is currently the most popular choice of activation function. One problem with the ReLU activation function is
that non-positive inputs result in a gradient of 0, meaning that a unit’s parameters may not be updated and "learn". Sometimes this results in a malignant learning phenomena know as the dying ReLU problem, whereby during training the activation of a neuron never exceeds 0 and the neuron’s parameters never receive an update. This is alleviated by a variant known as leaky ReLU, taking the form $f(x) = \max(ax, x)$, where $a < 1$ is some coefficient that determines the slope of the function in the negative domain.

The sigmoid activation function conveniently maps the input to the range $(0, 1)$. This may be useful in certain applications when values in this range are required, such as for percentages or probabilities. This range may lead to troubling learning dynamics, so a variant of the sigmoid function is sometimes used. This variant is the hyperbolic tangent function, sometimes called the tanh function, whose range is $(-1, 1)$. Both the sigmoid and the tanh function have horizontal asymptotes, and thus very long, flat tails. Outside a relatively small domain, the function is practically flat and the gradient approaches 0, which makes learning very sensitive to the magnitude of the input. Largely due to these tails, ReLU activation functions are preferred.
Computational Graphs

A neural network organizes operations into a graph structure called a **computational graph**. Specifically, a computational graph is a directed acyclic graph whose nodes correspond to operations and edges correspond to outgoing values. Variables, inputs, and other supplied values can be considered operations that have no indegree and simply return the supplied value. This structure is a natural and generic representation of a function, that is used to execute the function, as well as leveraged in order to find gradients on certain values (discussed in the section on automatic differentiation). These computational graphs have become the language of deep learning models, as seen in almost all frameworks such as in TensorFlow’s Graph, as well as in Caffe’s and ONNX’s schemas, among others.

A computation graph is a functional description of a particular function, for which there may be many functionally equivalent descriptions. For example, a fully connected layer may be discussed at many levels of granularity; in terms of layers as matrix multiplications, in terms of individual neurons as dot products, or even in terms of scalars. Another example is the sigmoid function, \( \sigma(x) = \frac{e^x}{e^x + 1} \), which may be implemented by specifying that a node implements the sigmoid operation, or by implementing it in terms of its constituent operations (exponential, division,
addition). This depends on the set of primitive operations available, e.g. if the sigmoid is not defined, one may need to implement in terms of primitives (exponential, division, addition).

A neural network may be considered a program that defines computational graphs. There are two types of neural networks: static and dynamic neural networks. A static neural network uses the same computational graph regardless of the input, whereas a dynamic neural network may produce distinct computational graph structures for distinct inputs.

**Operations**

As discussed in the previous subsection, a computational graph is composed of primitive operations. This subsection will introduce three important operations: the softmax operation, the convolution operation, and the recurrent neural network (RNN). Common operations include simple operations like dot products and addition, but may range to higher levels of abstraction like activation and layer functions and RNNs. Because higher level operations, like layer functions, may be implemented as a structure of low level operations, there is not an excellent distinction between a high level operation and a functionally equivalent structure of irreducible primitives like addition and multiplication. Although some operations, especially RNNs, are better understood
as architectures, they will be introduced as operations; they receive inputs and produce outputs.

The **softmax** function is a vector-valued function that maps from a vector to a probability distribution, \( \sigma : \mathbb{R}^d \rightarrow (0,1)^d \), such that the entries sum to one, \( \sum_i \sigma_i(\vec{v}) = 1 \). The softmax function is defined as:

\[
\sigma(\vec{v})_i = \frac{e^{\vec{v}_i}}{\sum_{j} e^{\vec{v}_j}} \quad \text{for } i = 1, \ldots, |\vec{v}|
\]  

(3)

The softmax function is often seen in classification tasks, where a neural network produces a probability distribution over classes.

The **convolution operation** is a special type of layer function. However, unlike most layer functions, the convolution operation must operate on tensors of (at least) rank \( \geq 3 \), and crucially, each neuron is reused several times. In essence, convolution takes several smaller "patches" from the input tensor, and runs each patch through each neuron. The outputs of each neuron constitute a "feature map", and the feature maps from all the neurons are stacked to form the output of convolution. In the context of convolution, neurons are referred to as filters. An illustration can be found in figure 3.

Each filter is swept through the input tensor, moving according to the **stride**, a hyperparameter to the convolution operation that determines how far to step along each dimension. At each position, a patch or "receptive field" is extracted.

\[^{3}\text{The rank of a tensor is its number of dimensions. For example, a scalar is a rank 0 tensor, a vector is rank 1, a matrix is rank 2, etc.}\]
from the input. Each filter takes the dot product between itself and the receptive field, appending it to the corresponding position of the output. Just as in fully connected layers, a bias and activation function are applied.

A popular implementation of convolution is to extract all $n$ receptive fields from the input, which are unraveled into row vectors and stacked vertically. Likewise, the $k$ filters are unraveled into column vectors and stacked horizontally. A matrix multiplication is then performed, and the bias vector, of length $k$, is added, resulting in an $n \times k$ matrix. This matrix is reshaped to the appropriate output shape, and possibly followed by an activation function.

It’s interesting to note that fully connected layers can be implemented as a convolution operation, by simply having each filter be the same shape as the input. This makes it so that the number of receptive fields is $n = 1$. In this case, each filter is acting as a typical neuron; it is connected to every neuron in the input, and produces a single output. Otherwise, a filter still behaves as a neuron, but is reused at $n$ regions of the input.

The convolution operation is very commonly used in computer vision tasks. Computer vision is peculiar because the important features of an image may appear throughout; important patterns may appear anywhere in an image. The convolution operation capitalizes on this assumption. It utilizes small feature detectors (filters)
that only look at small patches of the input, but are reused at multiple patches. This is in contrast to the fully connected layer, which is a large feature detector that looks at the entire image at once, or as a special case, each neuron is sparsely connected so as to only look at one small patch.

Parameter sharing/reuse is highly beneficial because it reduces the number of parameters to be learned. Consider the case of feature detection in an 1024x1024x3 image (the third dimension is the r,g,b color channels). A fully connected layer would require on the order of 1024x1024x3 ≈ 3 million parameters, whereas convolution would require on the order of 75 parameters for a filter that looks at 5x5x3 patches. In addition to improving memory and performance, parameter sharing benefits learning. Firstly, it reduces the dimensionality and sophistication of the problem. Second, it provides more opportunities for a feature detector to learn; it may experience hundreds of examples (patches) per image. Lastly, convolution allows filters to generalize; filters can detect features in regions where the feature has never before been observed, which is not possible with fully connected layers.

A recurrent neural network (RNN) is a family of architectures that contains some recurrence relation. This architecture is especially apt for sequential tasks, such as machine translation, speech recognition, and text generation. Although an RNN is best regarded as an architec-
nature, many popular libraries treat them as operations; they receive inputs and produce outputs.

Generally, an RNN repeatedly applies a function, $f : \theta, x_t, h_{t-1} \rightarrow (y_t, h_t)$ where $x_t$ is the $t^{th}$ element of the input sequence, $y_t$ is the $t^{th}$ element of the output sequence, $h_t$ is the $t^{th}$ **hidden state**, and $\theta$ are the parameters. This function $f$ is often called the RNN’s cell. By repeated application, an RNN maps from an input sequence, $x$, to an output sequence, $y$. The input sequence can be augmented arbitrarily, such as with begin- and end-of-sequence markers, or with zeros so that the RNN can continue to produce outputs. The output sequence can also be used arbitrarily; elements may be used as input to other operations, and some elements may even be ignored.

The hidden state of an RNN is an internal representation of the RNN’s world - any relevant information such as what input it has seen, what it has output in the past, previous "thoughts" it has had, etc. The initial hidden state, $h_0$, could be initialized to $0$, a sentinel, or decided by another network component. Further, the structure of the hidden state could be arbitrary; it could be a single vector, but also may be a more sophisticated object containing arbitrary tensors.

There are many variations within the RNN family. The original, and most simplistic, is often referred to as a "Vanilla RNN". This variant often performs poorly, due to an inability to handle long term dependencies, which is the charac-
teristic of being able to retain relevant information over many time steps. This is related to the "Vanishing Gradient Problem", a phenomenon whereby a learning algorithm is unable to effectively propagate a learning signal through multiple operations. There are two similar architectures that mitigate this problem: the Long Short Term Memory (LSTM), and the Gated Recurrent Unit (GRU). These circumvent the vanishing gradient problem by allowing the hidden state, and related gradient, to flow through time steps relatively unhindered. As such, these architectures unanimously outperform the vanilla RNN.

Lastly, there is a remarkable addition to an RNN called an attention mechanism. At each time step, a (usually large) subset of the entire input is combined using the previous hidden state. This combination is then input to the RNN. The combination function or attention mechanism varies for different tasks, such as audio, text, images, etc. If the attention mechanism is considered an internal component of the RNN, and the input elements are considered vectors of inputs, then the attention mechanism is consistent with the definition of an RNN.

Conceptually, RNNs are defined in terms of a recurrence relation or a loop structure. Although an RNN is conceptually cyclic, a neural network must be a directed acyclic graph. To reconcile, one can trace the execution of an RNN to yield the effective computational graph; an
acyclic graph of all the operations actually executed. This process is often called "unrolling" the RNN, and is demonstrated in figure 4.

One reason that RNNs are so effective is because they reuse the same parameters at every time step. Both RNNs and convolution use parameter sharing to reuse parameters through time and location, respectively. This effectively increases the number of experiences a parameter receives, and reduces the number of parameters in a model. Besides benefiting training in these respects, it also allows the model to generalize learned behavior to experiences that have never occurred at a given time/location, but may have been experienced at other times/locations. Parameter sharing in these cases also allows greater flexibility of a model; it may scale to inputs of variable size, in terms of sequence length for RNNs or tensor size for convolution.

In practice, neural networks are often defined in terms of many operations, some of the most popular being the fully connected layer function, softmax, convolution, and the RNN. Additionally, the cell function of an RNN is often defined manually. One implementation detail not covered in this subsection is that most systems’ operations are designed to operate on batches, meaning that they perform the operation on multiple input instances at once, which happens to be more computationally efficient.
Learning Algorithms

The Objective

The previous section described the structure of a neural network. This section will describe how parameters of a neural network are adjusted so as to improve the network’s performance on some task. The definitions of "performance" and "task" are defined by the practitioner, in terms of a scalar objective (e.g. score, accuracy, etc), and examples from the domain to be learned (e.g. classifying digits, driving a car). The process of learning by example is known as inductive learning, a paradigm whereby behaviors are induced given examples. This is in contrast to systems that employ deductive reasoning, e.g. expert systems, whereby behaviors are deduced by a process of entailment, given a model or set of true statements. Neural networks could thus be considered to be programmed by data, without the need for explicit models or rules, which may be unsuccessful or intractable to derive and write down.

Learning algorithms fall into two classes: those that fit example outputs and those that do not. Learning using example outputs is known as supervised learning, where the algorithm is generally presented with a set of input-output \((X, Y)\) pairs with the objective of having the neural network match the provided output given the input.
The function approximation capabilities of neural networks are also widely used for **unsupervised learning** tasks, which only take inputs without example outputs and aim to extract some hidden structure from the input distribution. In both cases, parameters are updated so as to optimize some objective, usually a scalar value called the "loss", $L$, to be minimized.

Formally, the parameters to be learned, $\theta$, will be separated out, and the neural network will be regarded as the function that takes parameters and an example and maps to a loss term. In the case of supervised learning, the network is formulated as $L : \theta, X, Y \rightarrow \mathbb{R}$, and in the case of unsupervised learning it is formulated as $L : \theta, X \rightarrow \mathbb{R}$. For the sake of generality, the abbreviation $L_i$ may refer to either $L(\theta, X_i, Y_i)$ or $L(\theta, X_i)$.

The loss term, $L$, is often the sum of two loss terms: the data loss and the regularization loss. The **data loss**, $L^d$, quantifies the performance of the network, for example in the supervised case, the disagreement between the network’s output and the desired output. The **regularization loss**, $L'$, is generally used to penalize the complexity of the model, for example the magnitude of the parameters. The regularization term is weighed by the **regularization strength**, $\lambda$, a term used to tradeoff between the data and regularization loss.
Altogether, the ultimate loss function is

\[ \mathcal{L}_i = \mathcal{L}^d_i + \lambda \mathcal{L}^r_i \]  

(4)

The goal of a learning algorithms is the following optimization problem:

\[ \text{find } \arg\min_{\theta} \mathbb{E}_i[\mathcal{L}_i] \]  

(5)

To solidify the concept of a loss, consider the case of a prototypical regression task. The neural network will be decomposed into two parts: a model function, which maps from inputs to outputs, and a loss function, which defines the "badness" of the model’s outputs. The model function is a function \( \Psi : \theta, X \rightarrow \hat{Y} \), and the loss function \( L : \hat{Y}, Y \rightarrow \mathbb{R} \) maps from the produced output \( \hat{Y} \) and the target output, \( Y \), to a single scalar term, the data loss. Thus the data loss function is a function \( \mathcal{L}^d(\theta, X, Y) = L(\Psi(\theta, X), Y) \). The regularization loss function is a function \( \mathcal{L}^r : \theta \rightarrow \mathbb{R} \). The ultimate loss function is thus

\[ \mathcal{L}(\theta, X, Y) = \mathcal{L}^d(\theta, X, Y) + \lambda \mathcal{L}^r(\theta) \]  

(6)

\[ = L(\Psi(\theta, X), Y) + \lambda R(\theta) \]  

(7)

Given a dataset \( D \) of \((X, Y)\) pairs, the optimization problem can be posed as

\[ \text{find } \arg\min_{\theta} \mathbb{E}_{X, Y \in D}[\mathcal{L}(\theta, X, Y)] \]  

(8)

by the properties of expectation, the regularization term is a constant and can commute outside
the expectation operator, so that

\[ E_{X,Y \in D}[\mathcal{L}(\theta, X, Y)] = E_{X,Y \in D}[L(\Psi(\theta, X), Y) + \lambda R(\theta)] \]

\[ = \lambda R(\theta) + E_{X,Y \in D}[L(\Psi(\theta, X), Y)] \]  

(9)

(10)

Now the term inside the expectation can be approximated by averaging over a sample of examples in the dataset, called a batch \( B \subseteq D \), such that

\[ E_{X,Y \in D}[L(\Psi(\theta, X), Y)] \approx \frac{1}{|B|} \sum_{X,Y \in B} L(\Psi(\theta, X), Y) \]

(11)

Finally, the objective could be optimized by iteratively updating the parameters, \( \theta \), so as to minimize the loss on the \( k^{th} \) iteration’s batch, \( B_k \):

\[ \lambda \mathcal{L}'(\theta) + \frac{1}{|B_k|} \sum_{X,Y \in B_k} L(\Psi(\theta, X), Y) \]  

(12)

Finally, once the parameters \( \theta \) have been optimized, relevant portions of the network can be extracted. Although in this example the network is composed of three functions, \( \Psi, L, \mathcal{L}' \), only the function \( \Psi \) is of interest, and the remaining functions can be discarded. In some cases, larger components of the network may be discarded, and relevant portions may be reused for other tasks like feature extraction, etc.
Learning Algorithms

There are various approaches to the aforementioned optimization problem. Some of the most naive approaches involve the use of random search. Hill climbing and simulated annealing are two related algorithms that iteratively improve a solution through random search. They both pick a neighbor of the current state, such as with a random perturbation, and if it is better than the previous state, it is accepted as the current state. The difference with simulated annealing is that it may accept a state that is worse, as per an acceptance probability that depends on how much worse the state is, and the annealing schedule. Although due to the vast search space (neural networks may have millions of parameters), both these methods are unreasonably inefficient.

Another approach is that of population-based methods, such as Genetic Algorithms and Natural Evolution Strategies. In some regards, Genetic Algorithms are a form of hill climbing, with the exception that candidates may not be in the neighborhood due to the crossover operator, as well as the selection/culling dynamics of the population. Natural Evolution Strategies takes a different approach. It parameterizes a distribution over the parameter space using a mean and covariance matrix, which it samples from to produce a population. It then evaluates the "fitness", I
or "goodness", of each individual in the population, which is used to refine the distribution mean and covariance. In effect, Natural Evolution Strategies uses a population in order to estimate a search gradient to follow. Natural Evolution Strategies has been shown to perform well given a reasonable amount of resources, and additionally has advantages like parallelizability and robustness. All of these techniques (hill climbing, simulated annealing, Genetic Algorithms, Natural Evolution Strategies) are rather unpopular in the field of neural networks, although they advantageously forego the calculation of gradients and thus the necessity for differentiable functions.

The most successful class of learning algorithms use the gradient to iteratively update parameters. **Gradient descent** (below) uses the gradient of the loss function, $\frac{\partial L}{\partial \theta}$, with respect to the parameters, $\theta$, to iteratively update parameters:

$$\theta_{i+1} = \theta_i - \alpha \frac{\partial L_i}{\partial \theta_i}$$

(13)

where $\alpha$ is a scalar or tensor valued term called the **learning rate**, sometimes called the step size.

As mentioned in the previous subsection, the objective $L_i$ is approximated by evaluating a batch of examples. The number of examples in a batch is referred to as the **batch size**. The effect of the batch size on learning is generally difficult to characterize from a theoretical perspective,
but it is understood that the larger the batch size, the closer the batch’s gradient is to the expected gradient. Although as the batch size increases, there are diminishing returns; a larger batch size may only slightly improve the accuracy of the gradient, and thus learning, but at a greater cost of computation. On the other hand, a smaller batch size may allow for more iterations, but the gradient may be too noisy. In practice, the practitioner must strike a tradeoff between speed and noise. Batch sizes like n=32, n=64, or n=128 are commonly used. Note that many dimensions, such as batch size, are conventionally powers of 2 because many backend systems and implementations simply work better with powers of 2.

For standard gradient descent, the choice of learning rate is crucial; it may slow down convergence if too low, or it may cause the algorithm to overshoot a good solution. Typically, like many other hyper-parameters, the learning rate is searched for either manually or through an automated process, e.g. cross validation. It may sometimes be helpful to schedule or anneal the learning rate, so that updates are larger earlier and smaller adjustments are made later. There are very compelling second-order methods, such as those based on Newton’s method of optimization, of the form:

\[
\theta_{i+1} = \theta_i - [H(L_i)]^{-1} \frac{\partial L_i}{\partial \theta_i} \tag{14}
\]

Where \([H(L_i)]^{-1}\) is the inverse of the **Hessian**
**matrix**, a square matrix of second order partial derivatives. Geometrically, this can be interpreted as scaling inverse to the curvature, so that smaller steps are taken in directions with more curvature, and larger steps taken with less curvature. Although this is ideal, the inverse Hessian is an infeasible term as-is because it requires the inverse of a giant square matrix; for a network with only one million parameters, the Hessian matrix would have a trillion entries. There is work being done on alleviating this problem, such as the BFGS algorithm (and limited-memory BFGS), although these methods are still considered infeasible and are rarely seen applied to neural networks.

There are popular **momentum based learning algorithms**, such as the momentum method and the Nesterov Accelerated Gradient method. The goal of these momentum methods is to incorporate a first-order force that somehow encourages a more direct path of descent, avoiding an oscillating or circuitous descent, as momentum does in the physical world. Both these methods involve an additional hyperparameter for the momentum magnitude, $\mu$, typically set to $\mu = 0.9$. The formula for the momentum method is:

$$v_{i+1} = \mu v_i + \alpha \frac{\partial L_i}{\partial \theta_i}$$  \hspace{1cm} (15)$$

$$\theta_{i+1} = \theta_i - v_{i+1}$$  \hspace{1cm} (16)$$

And the formula for the Nesterov Accelerated
Gradient method⁴ is:

\[
\theta_{i,a} = \theta_i - \mu v_i \tag{17}
\]

\[
v_{i+1} = \mu v_i + \alpha \frac{\partial \mathcal{L}(\theta_{i,a}, X_i, Y_i)}{\partial \theta_a} \tag{18}
\]

\[
\theta_{i+1} = \theta_i - v_{i+1} \tag{19}
\]

The difference between the two methods is that the Nesterov Accelerated Gradient method "looks ahead" to the position that the momentum update would send it, \( \theta_{i,a} \), and evaluates the gradient from there. It then applies the gradient update from \( \theta_{i,a} \). This "look ahead" is generally beneficial, and tends to outperform the original momentum method.

Because memory is often at a premium in neural network applications, it is worth mentioning that both methods require double the memory as compared to plain gradient descent. This is due to storing the previous parameter update, \( v_i \).

Lastly, there is a popular family of learning algorithms that use adaptive learning rates, called adaptive learning rate methods. This family includes Adagrad, Adadelta, RMSProp, and Adam, among other variants. Adagrad is an earlier method with the following update rule:

\[
c_{i+1} = c_i + \left( \frac{\partial \mathcal{L}_i}{\partial \theta_i} \right)^2 \tag{20}
\]

\[
\theta_{i+1} = \theta_i - \alpha \left( \frac{1}{\varepsilon + \sqrt{c_{i+1}}} \circ \frac{\partial \mathcal{L}_i}{\partial \theta_i} \right) \tag{21}
\]

⁴ Note that the formula for the Nesterov method uses \( \mathcal{L}(\theta_{i,a}, X_i, Y_i) \) rather than the shorthand \( \mathcal{L}_i \) because it evaluates \( \mathcal{L} \) at \( \theta_{i,a} \) rather than \( \theta_i \). The update for unsupervised learning would use the term \( \mathcal{L}(\theta_{i,a}, X_i) \).
Where $\circ$ is the hadamard (element-wise) product, and $\varepsilon$ is a smoothing term to avoid division by zero, usually $\varepsilon = 10^{-8}$. In essence, Adagrad decreases the learning rate along directions where the gradient is large, and increases where the gradient is small. This is similar to Newton’s Method or BFGS, with two key differences. First, the second derivatives are not explicitly computed. Second, although $c_i$ is a vector, it could equivalently be a diagonal matrix. This means that only the basis directions are being adapted, which in theory is not as good as a full Hessian matrix, but this simplifying assumption reduces the overhead from $n^2$ values to $n$. Similarly, using the inverse of the diagonal of the Hessian matrix in place of $c_i$ has been shown to work well. A significant property of Adagrad is that $c_i$ is monotonically non-increasing. In effect, this means that the learning rate will tend to 0, which may lead to premature convergence. Adadelta is a modification of Adagrad that addresses this issue by making $c_i$ be a running average.

RMSProp is very similar to Adadelta. The update rule is as follows:

\[
c_{i+1} = \gamma c_i + (1 - \gamma) \left( \frac{\partial L_i}{\partial \theta_i} \right)^2
\]

\[
\theta_{i+1} = \theta_i - \alpha \left( \frac{1}{\varepsilon + \sqrt{c_{i+1}}} \circ \frac{\partial L_i}{\partial \theta_i} \right)
\]

where $\gamma$ is the decay rate of the running average.
Finally, the Adam method is considered the state of the art learning algorithm for neural networks. Adam is similar to RMSProp, except that the parameter update uses a running average of the gradient rather than the gradient itself. This provides a better estimate of the expected gradient, as compared to the gradient at a single batch. The Adam update rule is:

\[
g_{i+1} = \beta_1 g_i + (1 - \beta_1) \frac{\partial L_i}{\partial \theta_i} \tag{24}
\]

\[
c_{i+1} = \beta_2 c_i + (1 - \beta_2) \left( \frac{\partial L_i}{\partial \theta_i} \right)^2 \tag{25}
\]

\[
\theta_{i+1} = \theta_i - \alpha \left( \frac{1}{\epsilon + \sqrt{c_{i+1}}} \circ g_{i+1} \right) \tag{26}
\]

Where \( \beta_1, \beta_2 \) are the decay rates for the gradient and learning rate running averages, respectively. These are typically set to \( \beta_1 = 0.9 \) and \( \beta_2 = 0.999 \) Not shown in the update rule above are additional components to the Adam algorithm used to compensate for the initialization of \( g_0, c_0 \), which are initially set to zero. This is discussed in section 3 of the original paper, entitled "Initialization Bias Correction".

For every parameter in the neural network, Adam must store a corresponding value in \( g_i \) and \( c_i \), thus requiring three times the storage as standard gradient descent. The momentum methods, Adagrad/Adadelta, and RMSProp all require twice the storage as standard gradient descent.
Regularization

Regularization is a term that may refer to a number of techniques used to avoid overfitting. Overfitting is the phenomenon whereby a model learns noise or peculiarities of the dataset rather than the underlying distribution. This is undesirable because an overfit model will not generalize well to unseen data. There are various forms of regularization, but only two will be covered here: dropout and weight regularization.

Briefly, dropout is an algorithm that randomly turns off or "drops out" neurons with a given probability during training. Dropout implicitly encourages robustness by forcing the model to perform well with incomplete information. The reason the authors believe it works well is that it prevents neurons from co-adapting such that neurons only perform well in the context of other specific neurons performing properly.

Weight regularization is a regularization method that is more theoretically grounded and understood. It adds a regularization loss to the total loss to be minimized, which is scaled by the regularization strength, $\lambda$, a term used to tradeoff between the data and regularization loss. To motivate the use of weight regularization, consider the case of a binary classifier. The binary classifier’s decision boundary can be considered a high dimensional manifold that splits the space in two; a positive and negative classification. Con-
cretely, the classifier $f(x)$ gives the distance to the boundary, which can be scaled arbitrarily by multiplying $f(x)$ by a constant. In effect, the distance to the boundary can be made arbitrarily large without changing the behavior of the model. By penalizing large weights, this problem is mitigated by inducing a preference for "smoother" boundaries, as demonstrated in figure 5. These "smooth" functions tend to generalize better than overfit functions. Another motivation is that by reducing the weights of a network, the network’s variance is reduced. Variance is a measure of the sensitivity of a model to slight perturbations in the training data.

There are various weight regularization functions, although only two of the major functions will be discussed: $L1$ and $L2$ regularization. $L1$ is calculated as:

$$L1(\theta_w) = \sum_{w \in \theta_w} |w|$$

(27)

And $L2$ is calculated as:

$$L2(\theta_w) = \sum_{w \in \theta_w} w^2$$

(28)

Where $\theta_w$ is the set of weight parameters. One notable difference between the functions is that the $L2$ function only depends on the total magnitude of the weight vector - it is a radial basis function.

A contrived, but informative, comparison of the functions is calculating the regularization loss.
incurred by the weight matrices $\theta_1 = \langle 0.5, 0.5 \rangle$ and $\theta_2 = \langle 1, 0 \rangle$. $L1(\theta_1) = L1(\theta_2) = 1$, whereas $L2(\theta_1) = 0.5$, and $L2(\theta_2) = 1$. As shown, $L2$ regularization prefers parameters with the more diffuse weights, whereas $L1$ is indifferent as to how weights are distributed. $L1$ often performs better on sparse problems, and $L2$ seems to perform better on more general purpose problems.

A more rigorous comparison of the $L1$ and $L2$ regularization is an analysis of their gradients:

$$\frac{\partial L1}{\partial \theta} = \begin{cases} 1 & \text{if } \theta > 0 \\ -1 & \text{if } \theta < 0 \end{cases} \quad (29)$$

$$\frac{\partial L2}{\partial \theta} = 2\theta \quad (30)$$

The magnitude of the gradient from the regularization loss w.r.t. the weights is constant for $L1$ regularization, and it is proportional to the magnitude of the weights for $L2$ regularization. This means that when the weights are small, $L2$’s gradient will be proportionately small and will push the weights towards 0 at a slow rate. Likewise, when the weights are large, $L2$’s gradient will be proportionately large and will push the weights at a fast rate. For $L1$ regularization, the gradient will be constant, regardless of the magnitude of the weights. So when the weight are small $L1$ will push the parameters to 0 at a faster rate than $L2$ would, and when the weights are large, $L1$ will push the parameters to 0 at a slower rate than $L2$ would.
In practice, the choice between regularizations is largely problem dependent and somewhat unpredictable. Both generally work well, and practitioners typically decide on a regularization based on empirical results. Perhaps the most important factor for regularization is the regularization strength; if the strength is too small, the regularization will be ineffective, and if it’s too large, it will inundate learning. Again, the best regularization strength is problem dependent and unpredictable, and is often chosen by manual or automatic hyperparameter tuning.

**Automatic Differentiation**

Automatic differentiation is the essential algorithm that enables gradient based learning algorithms for neural networks. In essence, automatic differentiation is a means to efficiently and conveniently calculate the gradients of an expression at a particular point in input space.

Note that this is not symbolic differentiation (*i.e.* finding a symbolic expression for the gradient), which may be humanly or computationally impractical. Nor is it numerical differentiation (*e.g.* finite differences), which requires a directional derivative to be calculated for each parameter, which for neural networks may easily entail millions of evaluations, whereas automatic differentiation only requires a single evaluation.

There are two types of automatic differentia-
tion: forward- and reverse-mode. Briefly, forward-mode allows for differentiation of many values with respect to a single value, whereas reverse-mode allows for differentiation of a single value with respect to many values. For neural networks, the goal is to differentiate a single value (the loss) with respect to many values (parameters), so only reverse-mode will be discussed and henceforth when automatic differentiation is mentioned assume it to be reverse-mode.

From a high level, automatic differentiation can be considered an algorithmic interpretation of the chain rule. The chain rule states that for a function composition \( R(x) = (f \circ g)(x) = f(g(x)) \), \( \frac{\partial R}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} \). This is sufficient for calculating gradients in the univariate case, when the computational graph is a tree (i.e. each node has no more than one input). Although to extend this method to arbitrary graphs, particularly those that are not trees, the chain rule must handle branches. The multivariate chain rule is a generalization of the univariate chain rule that states for a function \( R(x) = f(a(x), b(x)) \), \( \frac{\partial R}{\partial x} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial x} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial x} \), or more generally in vector notation, for \( R(x) = f(\vec{v}(x)) \), \( \frac{\partial R}{\partial x} = \frac{\partial R}{\partial \vec{v}} \cdot \frac{\partial \vec{v}}{\partial x} \).

For example, consider the function \( f(x) = e^x \times (x + 2) \). A diagram of the corresponding computational graph is shown in figure 6. Computers, as well as humans, evaluate operations one by one. One may evaluate \((x + 2)\), then evaluate \(e^x\), etc. But only so many ordering of operations are

![Figure 6](image_url)
legal; one may not evaluate an operation that has an undetermined input. Specifically, all edge relations must be preserved: if there is an edge from node $a$ to $b$, then $a$ must come before $b$ in the ordering, for all $a, b$. This implies a partial ordering over nodes, in which any possible linearization of the ordering is considered to be a topological sorting, of which there may be many. Further, the partial ordering can be formulated as a series-parallel partial ordering (figure 7), which describes how operations may be executed in parallel. This execution plan can capitalize on distributed or parallel environments, although for the sake of clarity, all discussion will be in terms of a linear topological sort.

A valid topological sorting can be used for staging operations. The output of each operation will be marked $s_1, s_2, ...$ where $s_n$ indicates the output of the $n^{th}$ operation. One such topological sorting for the function $f(x) = e^x \times (x + 2)$ would be:

\[
\begin{align*}
    s_1 &= x \\
    s_2 &= \exp(s_1) \\
    s_3 &= 2 \\
    s_4 &= \text{add}(s_1, s_3) \\
    s_5 &= \text{multiply}(s_2, s_4)
\end{align*}
\]

There are two phases to automatic differentiation: the forward pass followed by a backward pass. In the forward pass, the computational graph is traversed in topologically sorted order,
and operations are evaluated, as demonstrated above and in figure 8. As well as returning a result, each operation additionally stores any intermediary values needed for calculating gradients. These intermediary values are used for calculating the (local) gradient of the operation with respect to its inputs. It’s interesting to note that these local gradients can be calculated immediately upon executing the operation. For example, consider a multiplication operation that takes two scalar inputs, $x, y$. The derivative of the operation w.r.t $x$ is $y$, and likewise the derivative w.r.t. $y$ is $x$. An example of calculating local gradients is shown along the edges of figure 8. Pseudocode for the forward pass is presented in algorithm 1.

Algorithm 1: Forward Pass

**Input:** nodes

**Output:** output of each node, caches for each node (used for the backward pass)

sorted-nodes ← topological-sort(nodes)

outputs = {}
caches = {}

foreach node in sorted-nodes do

    inputs ← outputs[node["inputs"]]

    outputs[node], caches[node] ← node["forward"](inputs)

done

return outputs, caches

After the forward pass completes, the backward pass begins from the final operation that will be differentiated, which will be called $\mathcal{L}$. The
backward pass traverses the computational graph in reversed topologically sorted order, starting from $\mathcal{L}$. The gradient of the value between any node $a$ and any parent node (input) of $a$, $b$, can be calculated using the gradient of $\mathcal{L}$ w.r.t. $a$ and $a$’s local gradient w.r.t. $b$, which was cached in the forward pass. The gradient of $\mathcal{L}$ w.r.t. any node, $b$, can be calculated as the sum of gradients of $\mathcal{L}$ w.r.t to $b$’s outgoing edges, as per the multivariable chain rule. This is encapsulated in the following formula:

$$ \frac{\partial \mathcal{L}}{\partial b} = \sum_{a \in \text{child}(b)} \frac{\partial \mathcal{L}}{\partial a} \frac{\partial a}{\partial b} $$

Figure 9 demonstrates the backward pass for the computational graph presented in figure 8. The final operation in this figure, $\mathcal{L}$, is node $s_5$. Along each edge is the derivative of $\mathcal{L}$ w.r.t. the edge’s value, and inside each node is the derivative of $\mathcal{L}$ w.r.t. the node’s operation. Although this example uses scalar values and derivatives, the general methods still apply to tensor-valued computations and gradients. More general pseudocode for the backward pass can be found in
Algorithm 2: Backward Pass

Input: nodes, return-node, caches
Output: gradient of each node

grads ← {}
grads[return-node] ← 1
reverse-sorted-nodes ← reverse(topological-sort(nodes))
delete return-node from reverse-sorted-nodes

foreach node in reverse-sorted-nodes do
    grads[node] ← 0
end

foreach node in reverse-sorted-nodes do
    input-grads =
        node["backward"](grads[node],
        caches[node])
    foreach (parent, parent-grad) in
        collate(node["inputs"], input-grads) do
        | grads[parent] ← grads[parent] +
        | parent-grad
    end
end
return grads

It is shown that algorithm 2 yields the correct gradient by accumulating gradients in reverse topological order. By definition, in a topological ordering all outgoing edges of a node are traversed before it’s incoming edges, hence in reverse topological order, all of a node’s outgoing edges will be traversed before its incoming edges. Thus all of a node’s gradients on it’s outgoing
edges are summed before being propagated to its parent nodes, so that no term in the summation is missed and, by the multivariable chain rule, the gradient is indeed correct.
Historically, practitioners had to manually implement neural networks, gradient computation, and learning algorithms. Although this is tedious and error prone, it was not too problematic in the past due to the relative lack of data and number of operations. However, the requirements of neural network has changed. Modern "deep" neural networks often require very many operations and enormous datasets.

Modern networks are made tractable by the use of custom hardware and software. Neural networks typically run in distributed settings with specialized processors, such as GPUs or ASICs such as TPUs. Custom software is also used to accelerate computation, such as BLAS/LAPACK implementations and GPU/ASIC implementations written in a low-level interface such as CUDA. Altogether, these enable speedups of over 100x that of traditional implementations. Tooling has been built to conveniently implement neural networks and learning, saving practitioners time, bugs, and thousands of lines of rote operations and gradient calculations.
Neural network libraries and frameworks have emerged as the crucial means to develop modern neural networks. This chapter will discuss the principles behind many popular libraries and frameworks, including TensorFlow, Caffe, and ONNX. The focus of this chapter is not to outline details relevant to implementation, but rather high-level principles, themes, and contrasts.

**TensorFlow**

TensorFlow is a machine learning system developed by Google. TensorFlow is used to construct computational graphs, sometimes referred to as dataflow graphs. One fundamental distinction is that TensorFlow’s graphs may contain operations that explicitly handle the mutation and sharing of state, e.g. `tf.assign`. TensorFlow is a declarative system, but not purely functional.

TensorFlow has a "define-then-run" structure, implemented in two phases. In the first phase, the computation graph is defined via calls from the host language, such as Python or C++. Following the definition phase, the second phase executes an optimized version of the defined computation. Deferred execution is largely helpful for many optimizations. For one, source transformations and pruning can produce a computationally preferable graph. Second, one can better manage hardware, maximizing performance by staging and device placement. Despite benefiting
performance, deferred execution may introduce difficulties for users. Errors that arise during the execution phase may be difficult to trace. Additionally, it may be difficult to create a static graph for some tasks, e.g. working with arbitrary parse trees. A "define-by-run" variant of TensorFlow, TensorFlow Eager, was developed to support dynamic neural networks.

There are three types of tensors that are central to TensorFlow: Variable, Constant, Placeholder. Variables contain tensors that may be mutated with the \texttt{tf.assign} operation. Variables are often initialized to a value, or may be populated using \texttt{tf.assign}. In either case, a Variable must be initialized prior to its use. A Constant is an immutable tensor whose value is defined during the definition phase. Lastly, a Placeholder is a placeholder for a value that will be supplied when the graph is executed. A Placeholder could be considered an input argument to the computational graph; an undetermined value that will be supplied at runtime. A description of the Placeholder’s tensor must be supplied during the definition phase, including its data type and optionally its shape.

TensorFlow has a relatively standard treatment of computational graphs. The largest departures are that it is not purely functional, and that all values are considered tensors. All operations take $≥ 0$ tensors and return $≥ 0$ tensors. An operation’s behavior may be parameterized according
to its attributes; e.g. convolution has stride as an attribute. The setting of attributes varies by host language, e.g. the Python interface uses keyword arguments, but the internal TensorFlow representation, according to a node’s NodeDef, is an associative array. In the definition phase, outgoing edge values (outputs) are given descriptions, i.e. data type, tensor shape, etc. These descriptions allow TensorFlow to determine the behavior of the system, such as whether an operation is legal, memory requirements, etc, during the definition phase.

TensorFlow’s computational graph is represented as a Graph object, which can be mapped to a serializable GraphDef object that describes the Graph. The GraphDef contains multiple nodes, each represented as a NodeDef object. A NodeDef is an object with the following items:

- **name** - The node’s unique identifier, e.g. "node42".
- **op** - The name of the node’s operation, e.g. "MatMul" for matrix multiplication.
- **input** - A list of node names to serve as inputs, e.g. ["first-add", "node42"].
- **device** - The device the operation is executed on (for distributed settings), e.g. "GPU1".
- **attr** - An associative array of attributes that defines the behavior of the node’s operation, e.g. {"stride": 1}. 
Because a TensorFlow operation may produce multiple outputs, each node produces a list of outputs. For instance, a node with the name `foo` produces $n$ outputs where the $i^{th}$ output is referenced with the name `foo:i`. Because many operations only produce one output, by convention the name `foo` is synonymous with `foo:0`.

Finally, TensorFlow contains operations and scaffolding to support the training of models. The `tf.train` module contains various optimizers, including Standard Gradient Descent and the Adam Optimizer. It additionally contains queue implementations, which are useful for efficient preprocessing and management of devices. A typical training pipeline is shown in figure 10.

Variables are conveniently organized into collections, e.g. `TRAINABLE_VARIABLES`. Collections make organizing and handling specific subsets of Variables convenient; for example, optimizer operations will default to updating all Variables in the `TRAINABLE_VARIABLES` collection.
Caffe

Caffe is a neural network framework developed by the Berkeley Vision and Learning Center. Caffe is used to define neural networks as computational graphs, although unlike TensorFlow and other popular systems, a model is defined directly in a graph format, in a config file. There are bindings to programming languages, like Python, to aid in the construction of a config file, but config files are still the central construct of the framework.

Unlike TensorFlow, where learning algorithms and gradient updates are operations that mutate state, Caffe decouples the definition and mutation of a model into a model definition and a solver, both contained as files. The solver can train a model’s parameters by one of many standard gradient descent methods.

The atomic unit of computation in Caffe is referred to as a layer. Every network is composed of layers. A network typically begins with a data layer, an operation that loads data, and ends with a loss layer, a layer that defines the loss to be minimized. Perhaps the largest drawback of Caffe is that a network must be defined in terms of pre-defined layers, without access to primitive operations such as matrix multiplication. It is possible to implement custom layers, although this requires knowledge of C++, the format of Caffe’s data, called blobs, the ability to define

5 The most recent version of Caffe, Caffe2, contains matrix multiplication among other primitives.
the forward and backward pass, etc. The lack of primitive operations is a tremendous limitation of Caffe.

For the definition of a Caffe computational graph, each node is defined as an object consisting of:

- **name** - A unique identifier.
- **type** - The name of the layer (operation) to use.
- **bottom** - a list of input names, in order.
- **top** - a list of output names, in order.
- **param** - a list of parameter values to declare, e.g. weights or biases.
- **attribute** - an attribute may have a different name depending on the layer type, e.g. a Data layer uses the attribute name `data_param`. Similarly, the corresponding values depends on the layer type.

**ONNX**

ONNX (Open Neural Network Exchange) is a specification of a neural network in terms of a computational graph, operations, and data types. Various framework such as TensorFlow, Caffe, MXnet, etc use their own internal or intermediate representation of a static computational
graph. The virtuous goal of ONNX is to provide a common intermediate representation among several frameworks, enabling for interoperability between multiple packages, including Caffe, PyTorch, CNTK, Chainer, MXNet, and others. This allows one to transport a model to and from (import and export) supported frameworks.

Like Caffe, ONNX is file-centric and has an interface for constructing graphs via host languages like Python. An ONNX computational graph is structured as a list of nodes that must be in topologically sorted order. Similar to Caffe, each node is an objecting consisting of:

- **name** - A unique identifier
- **op_type** - The name of the operation to use.
- **input** - a list of input names, in order.
- **output** - a list of output names, in order
- **attributes** - a list of attribute-value pairs, specific to op_type.

Like Caffe, the definition of the network is decoupled from the learning algorithm. But rather than use a solver module, ONNX encourages one to train and deploy a network from any of the supported frameworks.

Lastly, much of the value of ONNX is in its rigorous specification. ONNX is replete with unit tests to ensure that backend libraries are faithful
to the ONNX standard. Practically, this allows for consistent transportation between libraries. These unit tests also provide a useful standard for operations, whose implementations may be equivocal otherwise.

**Summary**

This chapter reviews just a small fraction of notable neural network packages. Hitherto, this chapter has not discussed packages for dynamic neural networks, such as Pytorch, Chainer, and recently a variant of TensorFlow named TensorFlow Eager. These packages allow one to dynamically construct computational graphs using a "define-by-run" paradigm. This is in contrast to static neural networks, which are "define-then-run". Examples of static graphs are graphs created in TensorFlow, Caffe, and the ONNX specification.

Although the "define-by-run" paradigm is strictly more general and flexible than static definition, it must forego significant benefits and optimizations. Because dynamic neural networks lack the prescience of static neural networks, they may not appropriately plan on which values to store in memory, which device to store and execute operations on, etc. Further, modeling a dynamic neural network is an unreasonably complex task. Because a dynamic neural network may contain arbitrary control structures, one must effectively
model an entire programming language in order to encapsulate its behavior. Without modeling, it may be difficult to perform simple graph optimizations such as pruning. Additionally, the difficulty of modeling arbitrary control structures makes representations and standardizations like ONNX unrealistic.

An important distinction between frameworks is their API. The most popular form of API is via a host language like Python, a popular high-level language. An interesting deviation is Caffe’s API, which is primarily a graph language for defining models. Even though most packages define graphs from a host language, many construct and maintain an internal representation that is very similar to that of Caffe or ONNX. These representations effectively model static graphs, which is why they are typically used in a 2-stage paradigm where one defines a graph and separately executes it, such as in TensorFlow.

Conversely, dynamic neural networks are constructed on the fly, so that the definition and execution of a graph are integrated. This makes the API simpler for practitioners, who do not have to manage an enforced decoupling of definition and execution. Internally, a dynamic neural network package may create a familiar graph representation, like those seen in static graph packages, but will create a graph that is specific to an individual instantiation’s trace.

Almost all neural network packages contain

\[ \text{6} \text{ Like ONNX, Caffe contains an optional Python interface for defining graphs. Nevertheless, it is file-centric.} \]
primitive operations such as matrix addition and multiplication. Although neural networks are usually described at the level of abstraction of layers, most packages contain both layer functions as well as primitive operations like addition etc. This allows for the convenience of using common layers and operations, as well as the flexibility and completeness of creating custom layers and models by composing simpler primitive operations.
Taffy, a Compositional Representation

Compositional Nature of Neural Networks

Neural networks are understood in terms of high-level operations, ranging from simple layer functions to operations as sophisticated as residual blocks (figure 11) and Inception modules (figure 12). For this reason, Taffy generalizes a neural network to include abstractions.

The section on automatic differentiation described how gradients are found via a forward and backward pass of a computational graph, where each node is equipped with its own forward and backward function. From the perspective of automatic differentiation, a graph can be considered an operation; in the forward pass a graph takes inputs and produces outputs, and in the backward pass, a graph produces gradients on the inputs.

Similarly, an operation could be considered a graph. For instance, an operation may be composed of atomic or primitive operations, the most general being scalar operations. Further, there are many expressions that are functionally equiva-
lent to a given operation. Thus there is not an excellent distinction between operations and computational graphs. Central to the Taffy representation is the treatment of computational graphs as operations.

Henceforth, a computational graph may be referred to as a module. A neural network is a module consisting purely of primitive operations, or more abstractly, as a module composed of modules and primitive operations. Naturally, a module may reuse a particular module several times, e.g. a layer function may be reused several times in a model. Modules may also share structure, e.g. residual blocks and other modules may all utilize the same layer functions.

The interpretation of a neural network as a composition of modules is consistent with software development principles as well as the practice of developing neural networks. In software development, the "Don’t Repeat Yourself" principle roughly states that one should minimize the repetition of patterns by the use of abstractions. When one is developing a neural network from a host language like Python, one typically leverages abstractions like functions and classes. Ultimately, these abstractions are interpreted and transformed into functionally equivalent programs.

The same goes for modules; modules are interpreted via inline expansion and transformed into a module composed purely of primitive op-
erations. Because abstractions are compiled into primitives, abstraction does not provide any additional representational capacity. However, abstraction is tremendously useful for human interaction, cognition, and perception. Abstraction is what makes a good framework "good".
**Taffy Specification**

Taffy is a compositional representation of a neural network. This section provides an overview of the Taffy specification, from the top down. It will start with the specification of a library, the top-level container for modules. Modules will be discussed next, then the nodes that modules are composed of, and finally the primitive operations that nodes use.

**Library Specification**

Taffy represents a neural network as a module, a computational graph composed of inputs, primitive operations, and other modules. The top-level of a Taffy representation is a library. A library is an associative array, primarily containing a list of modules. An optional associative array may be used to store tensor values. Lastly, an optional value is provided for documentation. A library is an associative array defined in table 1.

Table 1: library spec

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modules</td>
<td>list of <strong>modules</strong></td>
<td>yes</td>
<td>an unordered list of <strong>module</strong> objects</td>
</tr>
<tr>
<td>tensors</td>
<td>associative array</td>
<td>no</td>
<td>name/tensor pairs</td>
</tr>
<tr>
<td>doc</td>
<td>free-form</td>
<td>no</td>
<td>documentation for the collection of modules</td>
</tr>
</tbody>
</table>
Module Specification

A module is at its simplest a computational graph. A node in a module may be either an operation or a placeholder. Placeholders are input values that will be provided upon instantiation. Some input values will be determined at runtime, whilst others are static and known at compile time. Operations may either be primitive operations or modules, which are treated as user-defined operations. Both operations and modules are documented using an op-doc object (short for operation documentation), defined in table 2.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>list of strings</td>
<td>strings documenting each input</td>
</tr>
<tr>
<td>output</td>
<td>list of strings</td>
<td>strings documenting each output</td>
</tr>
<tr>
<td>doc</td>
<td>free-form</td>
<td>supplementary documentation</td>
</tr>
</tbody>
</table>

Each module must have a name, a valid C identifier that is unique among both module names in the library as well as primitive operation identifiers. Modules must contain an input, an ordered list containing the names of placeholders in the module. Modules must contain an output, an ordered list containing references to values that will be returned from the module. Modules must contain nodes, an unordered list of node objects, which will be discussed in the next section. If a module uses another module as an operation, it
must explicitly import it by including it in an unordered list, \textit{module\_import}. If a module does not use other modules, \textit{module\_import} may be omitted. Note that module imports must be acyclic. An op-doc object, \textit{doc}, may be used to document the module.

In most other frameworks, operations may be parameterized by the use of an attribute (\texttt{attr}) whose values are known at compile time. Taffy parameterizes operations and modules using inputs, which are treated as first-class citizens.

A module is represented as an associative array, as defined in table 3.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>yes</td>
<td>a unique identifier</td>
</tr>
<tr>
<td>input</td>
<td>list of strings</td>
<td>yes</td>
<td>ordered list of input node names</td>
</tr>
<tr>
<td>output</td>
<td>list of strings</td>
<td>yes</td>
<td>ordered list of output values</td>
</tr>
<tr>
<td>nodes</td>
<td>list of nodes</td>
<td>yes</td>
<td>list of \texttt{node} objects</td>
</tr>
<tr>
<td>module_import</td>
<td>list of strings</td>
<td>no</td>
<td>module names to import as operations</td>
</tr>
<tr>
<td>doc</td>
<td>op-doc (table 2)</td>
<td>no</td>
<td>module’s documentation</td>
</tr>
</tbody>
</table>
**Node Specification**

A node represents every operation and input in a module. Each node takes \( m \geq 0 \) inputs and produces \( n \geq 0 \) outputs. Each node must have a name that is a valid C identifier that is unique among nodes in the same module. Each node must specify an operation to execute, \( op \). Each node must have input, a list of input values. A node may have an optional literal, a list of free form values that do not contain tensors. A node is represented as an associative array, as defined in table 4.

A value is referenced by the node name and output index; e.g., the second output value from a node with the name "mynode" is referenced as "mynode:1".

Table 4: node spec

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>yes</td>
<td>a unique identifier</td>
</tr>
<tr>
<td>op</td>
<td>string</td>
<td>yes</td>
<td>the operation’s identifier</td>
</tr>
<tr>
<td>input</td>
<td>list of strings</td>
<td>yes</td>
<td>an ordered list of input values</td>
</tr>
<tr>
<td>literal</td>
<td>list of literals</td>
<td>no</td>
<td>an ordered list of literals</td>
</tr>
</tbody>
</table>
Primitive Operation Definition

One of Taffy’s tenets is a uniform treatment between modules and operations. As such, there is a close resemblance between the definition and usage of modules and primitive operations. Like a module, every primitive has a \textit{op}, an identifier that is unique among primitives and modules.

Each primitive operation must specify its \textit{name}, one of 'tensor', 'control', 'placeholder'. Each primitive has a documentation object, \textit{doc}, as defined in table 2. Each primitive must also have a \textit{description} \textit{func}, a function that maps from inputs to outputs, evaluating all values except for tensors. Because a module is composed of primitives, defining \textit{description} \textit{funcs} for primitives implicitly defines \textit{description} \textit{funcs} for modules. This will be discussed in greater detail in the section on Taffy Puller’s Stage 2.

The definition of primitive operations is internal to the Taffy system, so users do not need to be aware of the internal representation. A description of the definition of a primitive operation is found in table 5.
Table 5: primitive operation definition

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>the operation’s identifier</td>
</tr>
<tr>
<td>type</td>
<td>string</td>
<td>the operation’s type</td>
</tr>
<tr>
<td>doc</td>
<td>op-doc (table 2)</td>
<td>operation’s documentation</td>
</tr>
<tr>
<td>description_func</td>
<td>javascript function</td>
<td>maps from inputs to outputs</td>
</tr>
</tbody>
</table>

**Primitive Operations**

This section will outline the core classes of primitive operations.

**Tensor Operations**

Tensor operations are central to neural networks and Taffy. Ultimately, tensor and placeholder operations are the only operations that remain after compilation. Each output of a tensor operation must be a tensor. Examples of tensor operations include matrix multiplication, element-wise addition and multiplication, the softmax function, etc. One exceptional tensor operation is the variable operation, which receives a tensor value and assigns a mutable tensor that initial value. Henceforth, every reference to the variable will reference the mutable value. Additionally, it will register that value in the library’s tensors.
**Placeholders**

A placeholder does not behave like a true operation; it is a placeholder for an unreceived value. A placeholder takes no input, and has a null description_func. Nodes that use the placeholder operation must be inputs in a module. When the module is instantiated, the placeholder will be substituted with an identity operation that forwards the supplied value.

**Control Operations**

Control operations are operations that are fully evaluated at compile time. They may receive arbitrary input values, such as tensors or objects, which may contain tensors. Control operation can not modify or produce tensor values, but may reference tensor values. Control operations may be control structures, or may construct and manipulate objects like data structures and functions. A control operation will be fully evaluated at compile time by evaluating its description_func. For example, a vector operation may pack its inputs into a vector. An important control structure is the identity operation, which takes $m$ inputs and produces $m$ identical outputs. Although control operations can not modify or produce tensor values, they may access tensor descriptions.

A node may contain an optional literal value

---

8 Although control operations may not produce new tensor values, they may reference new tensor values (called tensor descriptions), implicitly declaring new tensor operations. This is discussed in Stages 2 and 3 of the Taffy Puller.
that may be used as an input to the control operation. The rationale for having literal values separate from input values is that literal values do not depend on other nodes; they are fixed. This distinction is helpful for interpreting values as well as constructing dependency graphs.
Compiling Taffy

The Taffy compiler is decomposed into two separate components, the Taffy Puller and Taffy Packagers.

The Taffy Puller is a part of the Taffy compiler responsible for lowering the Taffy representation into a low-level intermediate representation. This intermediate representation is a flat (non-compositional) static graph that contains metadata such as value descriptions. Specifically, provided with a module to flatten, and a description of its inputs, the Taffy Puller will convert it to a representation comprised solely of tensor operations and placeholders, a representation that is apt for conversion to other representations and frameworks. There are three stages to the Taffy Puller:

1. Flatten and prune modules
2. Evaluating non-tensor operations and attaching value descriptions
3. Transformation and conversion to a low level intermediate representation

A Taffy Packager will take the intermediate representation produced by the Puller and compile it to other frameworks or specifications. For example, a Taffy Packager may convert the intermediate representation to TensorFlow code.
**Taffy Puller: Stage 1**

The first stage of the Taffy Puller is to flatten the compositional Taffy representation into a non-compositional representation comprised exclusively of primitive operations. Specifically, each module in the library is converted into a module that only contains primitives. This is achieved using an inline expansion procedure. Additionally, each module’s nodes are put in topologically sorted order, and nodes that do not contribute to the output are pruned.

First, the dependency graph of modules in the library is constructed. The dependency graph must be acyclic, otherwise there would be circular dependencies. Each module is then flattened in topologically sorted order. By flattening modules in topologically sorted order, each module will only reference modules that have already been flattened, as well as ensuring each module is only flattened once. A similar algorithm may flatten modules using recursion and dynamic programming.

The flattening of a module is a simple rewrite rule. To flatten a module, every appearance of a node whose operation is a module is replaced with the nodes from the referenced module. Recall that the referenced module has already been encountered, so that when integrating the module’s nodes, they don’t have to be flattened, sorted, and pruned again. Every integrated node
will be renamed using the module’s name. For example, if a node named "caller" uses a module that has a node named "node", that node will be integrated with the name "caller/node".

Each input placeholder will be replaced with an identity operation that receives the input value reference provided by the node’s input. Every value referenced in the node’s output is passed to an identity operation that has the same name as the node referencing the module. For example, if a node named "caller" uses a module, when the module is flattened the output will still be named "caller" and its outputs will still be referenced as "caller:0", "caller:1",... An illustration of a flattened module can be found in figure 13.
Figure 13: An illustration of Stage 1 of the Taffy Puller. The top figure shows a module to be flattened, utilizing another module in node "c". The middle figure shows the module referenced by "c", and the bottom figure shows a flattened representation of the top module. Note that the node in the bottom figure named "c" is an identity operation, yielding the same values as the node "c" in the top figure.
**Taffy Puller: Stage 2**

Stage 2 of the Taffy puller is responsible for evaluating a module. It is stipulated that the module’s inputs and outputs must be tensors. Stage 2 begins with a module, as well as descriptions of the input tensors, in terms of their shape and data type. Stage two evaluates the module, producing a trace of values produced by each operation, as well as a trace of tensor values. These traces serve two purposes: the trace of values helps in the analysis of the module and the behavior of each operations, and the trace of the tensors is used in stage 3 to construct a computational graph that is composed solely of tensor operations in a more constrained format.

Many system, like TensorFlow, execute a program from a host language to define a computational graph, which is extracted from the trace of tensors. Stage 2 works similarly, keeping track of the values produced by each operation, much like a host language would, but also keeping track of tensor values and how they’re produced. All values are fully evaluated during stage 2, besides tensors. Rather than produce actual tensor values, the Taffy puller produces **tensor descriptions**, defined in table . The tensor trace is then an associative array with references (valRefs) as keys and tensor descriptions as values. A **value description** is any value, that may be (or contain) a tensor description. For example, a value descrip-
A compositional representation may be an object like a list of tensor descriptions, but may also be another object or even a literal like a number. The value trace is simply an associative array with references (val_refs) as keys and value descriptions as values, as seen in table.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape</td>
<td>tensor shape</td>
<td>tensor’s expected tensor shape(^9)</td>
</tr>
<tr>
<td>data_type</td>
<td>string</td>
<td>tensor’s data type</td>
</tr>
<tr>
<td>val_ref</td>
<td>string</td>
<td>reference to the tensor value</td>
</tr>
<tr>
<td>op</td>
<td>string</td>
<td>identifier of the operation that produced the tensor</td>
</tr>
<tr>
<td>input</td>
<td>list of strings</td>
<td>list of references to input tensor values</td>
</tr>
<tr>
<td>attr</td>
<td>object</td>
<td>object that parameterizes the operation</td>
</tr>
</tbody>
</table>

For example, consider a series of nodes \(a, b, c, d, e\). Let node \(a\) be a tensor operation that produces a new tensor, \(a:0\). Let node \(b\) be switch statements that happens to route the value produced by \(a:0\). Node \(c\) may produce a list of items that includes the value routed from \(b\) as the \(8^{th}\) item. Another node \(d\) may unpack the list from \(c\). Lastly, node \(e\) will produce a new tensor value, producing a new tensor description. This example is illustrated in table.

\(^9\) A tensor shape is a vector describing the shape of a tensor. Entries in a tensor shape vector must either be positive integers or strings that are valid C identifiers.
In some regards, stage two is restrictive; it only allows for control structures that can be evaluated at compile time, which enables the Taffy Puller to form a static graph whose behavior can be described at compile time. Another possibility is to pass over Stage 2, compiling directly to a host language that could handle arbitrary control structure, enabling the definition of dynamic neural networks. Although the proposed framework, up to Stage 2, would allow for a complete, purely functional programming language, the Taffy Puller will use Stage 2 due to the prevalence of static neural network and the affordances of knowing behavior at compile time.
Stage 3 of the Taffy Puller is responsible for transforming the graph to only include tensor and placeholder operations. The intermediate representation produced by Stage 3 represents a module as a list of nodes, in topologically sorted order. This intermediate representation is a simplified representation called **Unwrapped Taffy**. In Unwrapped Taffy, each node has a unique name, an operation identifier, attribute object, and inputs (a list of value references), as defined in table . This resembles TensorFlow, Caffe, and ONNX’s intermediate representations.

Recall that the outputs of the module must be tensor values. The construction of the intermediate representation begins with these values: the tensor descriptions reference other tensor descriptions. From every tensor description, the input tensor descriptions are added to the search queue, etc. This search yields an unordered list of tensors (and associated tensor operations) that the module’s outputs depend on.

The tensor descriptions are mapped to a list of node objects, which are put in topologically sorted order. Node objects in Unwrapped Taffy are defined in table .
Stage 3 extracts nodes from the tensor trace produced during Stage 2. Because nodes are extracted from the execution trace, the structure of the module may change during Stage 3. In order for a tensor operation to make it into the stage 3 module, one of its outputs must be an ancestor of an output value. Not only can nodes be pruned, but further, new nodes can be introduced by reference from a tensor description. Although unusual, a control operation may produce a new node (that uses a tensor operation) by creating a new tensor description.
Taffy Packagers

A Taffy Packager converts the simplified representation produced by the Taffy Puller (known as Unwrapped Taffy) to a representation or program that can be used by another system. This allows for neural networks using the Taffy representation to be transported to other backends. Taffy Packagers may also allow for learned tensor values to be brought back into the Taffy representation, allowing for training to take place on one backend and the results transported to another.

Different Taffy Packagers must be written for different systems. Converting from Unwrapped Taffy to ONNX or Caffe is rather simple as they all use a similar graph representation. Converting to TensorFlow or TensorFlow.js is achieved by code generation. In all cases, the behavior of operations must be consistent among backend systems. As such, each tensor operation must define its standard behavior, which all Taffy Packagers must comply with. A tensor’s description_func must also be consistent with this definition.
TaffyStudio

TaffyStudio is an interactive visual interface to Taffy, allowing users to create and compile modules while providing continuous visualization, annotation, and debugging. The synergy of these features produces a novel system that is radically easier to use.

User Interface

Creating a Module: Name and Imports

Adding a new module and defining its name and imports can be done from the navigation bar, as shown in figure 14.

A new module can be created by clicking the "plus" button on the navigation bar. This will create a new tab for that module. This tab is not only used to navigate to the module’s nodes, but can also be used to change the module’s name and imports. The module’s name can be changed by double clicking the module’s name.

One can change a module’s imports by clicking the imports button, which gives collections of...
modules that can or have been imported. Available modules are selected so as to avoid circular dependencies. A module can be moved between these collections by clicking on the module.

Creating a Module: Defining Nodes

A module’s nodes, inputs and outputs can be defined from a module’s nodes panel. A node can be created by clicking on the panel, which will create the node and prompt a "sidebar" interface.

Figure 15 shows a node. The black circles (ports) on the bottom represent a node’s inputs, and the ports on top represent its outputs. The text inside the node is the name of the operation it uses, or the name of node if its provided by the user. To accommodate variadic functions, the sidebar allows one to change the number of input ports.

To populate a node’s inputs and outputs, the user draws edges between an input and output port. An edge may be deleted by clicking on it. As will be described in the debugging section, hovering over an edge will display the value flowing along it.

The sidebar, shown in Figure 16, allows users to populate a node’s attributes such as its name and operation. The sidebar takes on different functionalities for different nodes. For the input and output nodes, the sidebar allows one to change the number of inputs and outputs, populate the input descriptions, and compile the module.
For other nodes, the sidebar displays and allows one to edit the node’s attributes such as its name and operation. Once a node’s operation is defined, the sidebar will display the operation’s documentation, and possibly an interface for defining the node’s internal literals.

Many modifications to the sidebar will change the module in a way that will prompt an evaluation, which may update aspects of the GUI such as the ports and the values attached to certain elements.

**Creating a Module: Evaluation and Debugging**

Evaluations are performed to update values and debug errors in the graph.

When a user hovers over a port or edge, a tooltip displays the value flowing along it, as shown in figure 17. This makes tensor operations easier to construct because the shape and type of the tensor arguments are readily accessible. Arbitrary operations also benefit because the form and value of the arguments are queryable.

If an operation fails, it will throw an error with a message describing the error. This will appear as a warning attached to the node, as shown in figure 18. To help identify the cause of the error, one can inspect the values provided to the node. To support nested debugging, warnings may include links to the next module in the call graph. Values are extracted from the stack trace and
attached to the nodes in the modules involved, reflecting the state at the time of the error.

**Inputs, Outputs: Documenting a Module**

Modules and primitive operations are documented in the same format, in terms of its inputs, outputs, and an overview description. When a user selects a module or primitive operation, its documentation appears in the sidebar, as in figure 16.

Users can edit a module’s documentation by clicking on the module’s input or output node. This interface is displayed in the sidebar, as shown in figure 19.

**Inputs, Outputs: Input Descriptions**

When compiling a module, descriptions of the inputs must be provided to evaluate control flow and tensor operations. The interface for editing a module’s input descriptions is in the sidebar, after clicking the module’s input node, as shown in figure 20.

Taffy only compiles modules whose input and output are tensors, however modules used by other modules may take and emit arbitrary values. Hence, TaffyStudio’s input descriptions are not restricted to tensor descriptions, but may also be JSON objects, as shown in figure 20.
Implementation

TaffyStudio is implemented in slightly more than 2000 lines of functional JavaScript, which are packaged into a standalone file that can be imported into a webpage. TaffyStudio can be embedded into an HTML element with a single function call. TaffyStudio depends on the following libraries:

- **Taffy** for annotating, debugging, and compiling modules
- **D3** for manipulating SVG and facilitating interactions
- **Dagre** for automatic graph layout of imported modules
- **Bootstrap** for CSS and icons

Evaluating a Module

When a module is changed, TaffyStudio automatically commits those changes and prompts an evaluation. To evaluate a module, TaffyStudio organizes the user’s modules into a library which is passed to the Taffy puller, resulting in a value trace. Values in the value trace are attached to node elements in the graph; when a user hovers over a port or edge, a tooltip formats and displays the corresponding value.
In the event of an error, Taffy passes a partial value trace along with an error message and the node where the error occurred. TaffyStudio usually only attaches values to nodes in the module being compiled, however to enable annotations during nested debugging, TaffyStudio attaches values to all modules involved in the error. TaffyStudio traverses the path from the node where the error occurred, annotating values present at the time of the error.

TaffyStudio transforms a module before sending it to the Taffy Puller. This is required because input descriptions and outputs may not be tensor descriptions. Further, when a module is being developed its output may be undefined. Hence TaffyStudio transforms and pulls the module, then inverts the effect of the transformation on the value trace.

**Importing Modules**

Most actions executed in the GUI have a corresponding action that can be called and parameterized programatically. These "no-GUI" actions are used when importing a module, such as when importing the standard base modules. When importing modules, TaffyStudio uses the Dagre graph layout library to position nodes in the graph.
Case Study

To illustrate the benefits of TaffyStudio, this section contrasts implementations in TaffyStudio against corresponding implementations in popular frameworks. All implementations were chosen to be concise and representative of the best implementations available.

Dense Layer

Constructing a dense layer is a rather low-level task that most users will not do themselves. However this is an interesting task because it involves a significant amount of non-tensor operations. In many ways this task is more suitable for a programming language like Python.

An attempt was made to display TensorFlow’s implementation of a dense layer, however this proved difficult as their implementation straddles many source files, spanning both the TensorFlow and Keras libraries. As such, this section will compare against a custom dense layer written in Python with TensorFlow. Note that the "get_weight" and "get_bias" functions and modules are omitted from these graphics, but a similar story applies.

Figure 21 shows the TensorFlow implementation, and figure 22 shows the TaffyStudio implementation.
The TensorFlow implementation has the user annotate operations with a comment that gives the shape of the results, e.g. "(784, activations)". This has many drawbacks. First, the user must annotate the operations, which is tedious and may be done incorrectly. Errors in annotation may compound as later decisions may depend on these annotations. Correctly annotating becomes more difficult when using functions like "get_weight", where the internal behavior is not obvious, or when operations sophisticated semantics, like convolution.

TaffyStudio automatically provides annotations which can be accessed by hovering over values, as shown in figure 22. These annotations are not only correct, but are more informative as they also display the value’s data type and size. These annotations are only displayed when hovered over, removing clutter from the environment.

This comparison demonstrates how non-tensor computation is integrated into Taffy. The first 5 lines of figure 21 are devoted to ingesting the input, by way of giving a default value to an ar-

```python
def dense_layer(tensor, activations=None):
  if not (tensor.shape and len(tensor.shape)==2):
    raise ValueError("First input must be tensor of rank>=2")
  activations = tensor.shape[1] if activations is None else activations
  if not (isinstance(activations, int) and activations>0):
    raise ValueError("Second input, activations, must be a positive integer.")
  weight = get_weight(tensor, activations) # (784, activations)
  XW = tf.linalg.matmul(tensor, XW) # {batch, activations}
  bias = get_bias(XW) # {activations}
  XW_b = XW - bias # {batch, activations}
  return XW_b
```

Figure 21: A TensorFlow implementation of a dense layer.

Figure 22: A TaffyStudio implementation of a dense layer.

Figure 23: An annotation.

Figure 24: JavaScript code for the "ingest" operation.
argument and asserting certain conditions are met. This same functionality is encapsulated in figure 22’s "ingest" node. The ingest node, shown in figure 24, implements the ingestion functionality using the "js_function" operation, which takes JavaScript code as a literal. The flexibility of description functions allows Taffy to effectively mix traditional text-based programming with graphical programming, enabling complex functionality in lower-level modules.

ResNet

A ResNet architecture is a popular Neural Network architecture composed of so-called residual blocks. Residual blocks are themselves composed of mid-level operations such as convolution and batch-normalization. Hence each residual block is rather high-powered in itself, containing many parameters and operations.

The architecture of a residual block is shown in figure 25. The input is fed through a stack of operations, the result of which is added to the input. This architecture is said to have a "skip connection", a pattern which has shown to benefit the training of Networks.

This structure is clearly visualized in TaffyStudio, but is not obviously visible in the TensorFlow code of figure 26. In order to see that the result is added to input, one must notice the "+" in the "+=" operator. Despite this being well-written
code, the structure can only be understood with careful inspection.

A characteristic of the residual block is that every operation preserves the shape and data type of its incoming tensors. This property means that the result will be the same shape and data type as the input, so that they can be added without broadcasting. In TaffyStudio, this property is observable by hovering over a value, which gives its shape and data type. This is not only helpful in understanding a module, but also when building a module that is supposed to have this property.

This property is not evident in the TensorFlow code. Foremost, theses shapes are determined by arguments that are not yet present, e.g. "filters". TaffyStudio avoids this issue by requiring input descriptions, allowing a module to be inspected and debugged during construction. Even if arguments were present, the semantics of the convolution operation are complex, making the resulting shape non-obvious. Even if these semantics were obvious, there are no comments annotating shapes in the graph.

This information is clearly presented in TaffyStudio. Graph visualization is an intrinsic part of the environment, and the properties of the graph, specifically the shape and data types, are readily available as queryable annotations.
Conclusion

This work presents Taffy and TaffyStudio, a specialized representation and integrated development environment for constructing Neural Networks. Taffy and TaffyStudio are novel tools that greatly differ from today’s status-quo.

Taffy defines a Neural Network topology using a graph data structure, rather than a programming language. Taffy allows nodes to run arbitrary computations, allowing users to mix text-based programming with graph-based computation. Taffy emits an intermediate representation that can easily be compiled to and mixed into arbitrary backends.

Central to Taffy is the module abstraction, a composable graph akin to a user-defined function. Modules allow Networks to be constructed and interpreted at many levels of abstraction, making it simpler and easier to create Networks. Modules naturally group operations into a hierarchical structure, aiding visualization and debugging.

TaffyStudio is an interactive visual interface to Taffy, which strives to be easy to learn and use.
By developing a specialized integrated development environment, TaffyStudio vastly improves on weaknesses of other systems, especially visualization, debugging, and annotation.

Rather than visualize a Network after it’s been compiled, visualization is the primary method of construction. Instead of having programmers manually annotate code, TaffyStudio automatically annotates a Network, helping users create operations and debug errors.

TaffyStudio is tightly coupled with Taffy, continuously evaluating and debugging modules as they’re being built. Continuous evaluation allows TaffyStudio to annotate values, helping users inspect and debug a module as it’s being defined. When errors occur, TaffyStudio freezes execution and provides helpful interactive debugging, allowing users to drill through nested structures and inspect values at the time of failure.

Together, these features make for a system that’s substantially easier to use.

**Future Directions**

This section proposes future use cases for Taffy and TaffyStudio.

**End to End Experience**

TaffyStudio allows users to build a Neural Network model, but does not aid in the training
and deployment. Unfortunately, these necessary steps still require users to write complicated and monotonous code.

An extremely promising direction for TaffyStudio is integrating it into a platform that enables an end to end experience by allowing users to build, train and deploy Neural Network models with minimal effort and complexity. This platform could be implemented on top of a cloud-service provider, where a user can build new models, launch training jobs, and deploy the model to be served from an API. TaffyStudio effectively solves the problem of building Networks, but does not address the training and deployment components.

Currently, deploying and serving a model is relatively straightforward, and so this process could be automated relatively easily. Training a model remains challenging, involving loading, cleaning, featurizing and transforming data into a usable form. Further, efficiently optimizing hyperparameters, validating models, performing fine-tuning and transfer learning are some important tasks that remain non-trivial barriers to a simple yet easy training solution. Nonetheless, it seems that an end to end platform can be made to support a large class of use cases.
Integration into Development Environments

There exist a number of popular browser-based development environments. Jupyter Notebooks is perhaps the best known environment, which has been adopted in products such as Google’s Colaboratory and Amazon’s SageMaker. These environments are an excellent fit because they primarily target data science related tasks, often involving the development of Neural Networks.

TaffyStudio was built to be extensible and integratable, allowing a session to be embedded anywhere with just a single function call. With slight modifications to where compiled code is routed and how state is saved, TaffyStudio could very easily be integrated into notebooks.

Applying to other Domains

Taffy and TaffyStudio were made for building Neural Networks in a graphical format, however this tooling and approach could readily be applied to other domains. Operations could be added to Taffy and domain-specific features added to TaffyStudio to support the construction of various programs, for example SQL queries, data mashups, and pipelines.
Future Work

Control Flow Operators

One tradeoff Taffy made was to compile modules to an intermediate representation that only contains tensor operations. Unfortunately, this precludes many important control flow structures, making certain dynamic tasks impossible, for example applying an RNN over a dynamically shaped tensor.

Fortunately, Taffy’s generated code can be mixed with conventional Neural Network code, allowing for these capabilities. In the future, Taffy may natively support dynamic structure by allowing for constrained control flow operations, like looping. Another option is to compile to a more complex intermediate representation that supports dynamic execution.

Failsoft Operations and Suggestions

Primitive operations may fail if invalid parameters are provided. Some operations may reconcile invalid or incomplete arguments by using default arguments or broadcasting tensors, for example. Taking from the Scratch programming language, Taffy could further these efforts to try to eliminate even more errors. To this end, a balance must be struck between reducing errors and providing clear and sensible behavior that’s faithful
to the user’s intentions.

Currently, Taffy only allows operations to send messages to the user by using JavaScript’s try/throw interface. Unfortunately, throwing a message is lethal, barring operations from making suggestions without causing compilation to fail. Ideally, Taffy would expose an interface to throw non-critical suggestions or warnings.

**Variable Reuse**

Variable reuse is a common technique with many applications such as Recurrent and Siamese Neural Networks. Most popular systems allow users to reuse variables by attaching variables to functional objects which are reused.

Taffy supports variable reuse by explicitly passing variables. Unfortunately this does not scale well to modules with many variables, which may have to package/unpackage variable bundles using the "pack_list" and "unpack_list" operations. There are a number of solutions to facilitate variable reuse in Taffy:

- Expose a compiler directive that modifies the "get_tensor" operation’s behavior according to its scope, similar to TensorFlow’s "reuse" directive

- Define an operation to compile modules to a reusable functional object, in the fashion of Keras layers
Improving Graph Layout

TaffyStudio could improve on two aspects of graph layout: edge layout and automatic graph layout. Currently, edges are drawn as second order bezier curves, without regard to the remainder of the graph layout. Ideally, edges would avoid other nodes, bundle with related edges, and generally promote clarity. Second, TaffyStudio’s automatic graph layout algorithm could improve for small and large graphs.

Type System, Refining Ports

Taffy’s modules and primitive operations have documentation objects that assign documentation strings to a module’s inputs and outputs. Adding types could improve documentation and the TaffyStudio interface, specifically the ports interface.

Most other visual programming environments, like Scratch and Blender Nodes, stylize a node’s ports according to the value it accepts. In TaffyStudio, edge ports are not stylized, a simple improvement that could benefit interaction and visualization.

Taffy Modules Manager

Many other programming languages have package managers for importing code. TaffyStudio has functionality for loading Taffy modules, but
could benefit from an explicit visual interface for importing modules. Further, having a central store could allow for a community to develop and share new modules.

**Hyperparameter Registration**

Hyperparameters are one of the more fickle aspects of training a machine learning model. When training a model, practitioners will adjust these parameters either manually or programatically, using some form of search or hyperparameter optimization method. Unfortunately, some of these hyperparameters are baked into the compiled Taffy module, making it difficult to change hyperparameters without changing and recompiling the Taffy.

One solution would be to allow users to register literals as hyperparameters, which can later be modified in the compiled module.
Bibliography


Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick,


Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, and Jürgen Schmidhuber. Natural evolution strategies, 2011.