My name is Lisa Anne Hendricks and I am broadly interested in computer vision and machine learning. Currently I am interested in various vision tasks for video, such as activity recognition or automatically generating video captions. Specifically, I have applied one class of neural networks, recurrent neural networks, to video in order to learn long term temporal dynamics in video. I would like to be able to apply parallel computing to problems I might face in computer vision and gain a better understanding of how tools and software I actively use (e.g. Caffe - http://caffe.berkeleyvision.org/) are parallelized.

Deep neural networks have transformed computer vision by drastically increasing performance on a variety of vision tasks, such as image classification and detection. Deep neural networks are composed of multiple layers which each receive input from the previous layer. The output of each layer is computed as $y = G(f(x))$ where $f(x)$ is some linear function of $x$ (such as convolution in a convolutional neural network) and $G$ is a nonlinearity, such as a sigmoid or hyperbolic tangent. The parameters required to compute the output are learned by backpropagation, which updates the weights in each layer by computing the error between the desired output and computed output and propagating this error backwards through the network. Networks that have the capacity to model inputs with the wide variation seen in large datasets such as ImageNet[1], require millions of parameters (Krizhevsky et al. [2]).

GPU implementation is essential for fast training and testing of neural networks. For example, when training a network using the open source neural net implementation Caffe [3], train and test time decrease dramatically when using a GPU and the cuda BLAS library. The plot below [4] (image taken from [4]) shows an 11x speed up in computation time when using a GPU as opposed to a CPU implementation. Note that when using hardware specifically designed by NVIDIA for deep networks, a 14x speed up is seen [5].

Though a single GPU can greatly accelerate neural network training and testing, sometimes it is desirable for multiple GPUs to be used at once because the neural network model and data may not fit onto a single GPU or training with one GPU will simply take too long. Many researchers have approached the question of how to best use multiple GPUs to train deep networks more effectively, and I discuss one such method to parallelize convolutional neural networks described by Krizhevsky in his article ‘One weird trick for parallelizing convolutional neural networks’ [6]. Many convolutional neural networks comprise both convolutional layers, where the linear function $f(x)$ is a convolution of the input and a set of learned weights, and fully connected layers, where the linear function $f(x)$ is an inner product between the input and a set
of learned weights. Krizhevsky’s key insight is that different types of parallelization are better for different types of layers. Convolutional layers generally require a large amount of computation, but have fewer parameters than fully connected layers. Thus, sharing updated parameters between GPUs is less burdensome than for fully connected layers, and splitting data amongst multiple GPUs helps speed computation. To this end, Krizhevsky proposed parallelizing the convolutions with ‘data parallelization’, where the same convolutional layers are on each GPU, but the GPU process different batches of the data, and parallelizing the fully connected layers with ‘model parallelism’, where the model is split amongst GPUs, but each GPU sees the same data.

For data to pass from the convolutional layers to the fully connected layers, the system could naively place all outputs from the convolutional layers into a single batch and pass this large batch into the GPUs which have the fully connected layers. This, however, requires training to stop in order to aggregate data. Furthermore, GPUs have limited memory so a large batch of data might be undesirable. Alternatively, the tasks for running the data through the fully connected layers and passing the data from the convolutional GPUs to the fully connected GPUs can be pipelined; as the GPUs which process the fully connected layers process data from the first convolutional GPU, the next convolutional GPU can begin passing data to the fully connected GPUs. Krizevsky reports that when using 2, 4, and 8 GPUs respectively, he is able to accelerate training by 1.95x, 3.74x, and 6.25x with no loss in accuracy. This outperforms other methods which see a 2.2x speedup using 4 GPUs [6] and a 3.2x speedup using 8 GPUs [7].

Krizevsky’s insight could be applied to other neural net models, including the particular model I have recently worked with for video classification [8]. In my model, a set of \( N \) video frames serve as inputs to \( N \) convolutional networks. The outputs of each convolutional layer are fed into a recurrent neural network (specifically a long short term memory network). This network learns a set of weights that transforms an at time \( t \) and a hidden state at time \( t-1 \) to a hidden state at time \( t \). In order to parallelize my network, I could use data parallelism for the \( N \) convolutional layers and model parallelism for the recurrent layer. Thus far, my model training has been severely limited because I can only fit a relatively small amount of video data onto a single GPU (~24 clips of length 16; for reference, when training models on images, the batch size is generally ~128) and it trains fairly slowly (~3 days) making it difficult to explore various tweaks to the model architecture.


