A Convolutional Neural Network For Cosmic String Detection in CMB Temperature Maps

Razvan Ciuca, Oscar F. Hernández, Michael Wolman

Abstract. We present the convolutional neural network used in reference [11] to detect cosmic strings in Cosmic Microwave Background (CMB) temperature anisotropy maps. By training our neural network on numerically generated CMB temperature maps with and without cosmic strings the network can produce prediction maps that locate the position of the cosmic strings and provide a probabilistic estimate of the value of the string tension $G\mu$. The network as applied to noiseless simulations of CMB maps with arcminute resolution was able to locate strings and accurately determine the value of the string tension for sky maps having strings with string tension as low as $G\mu = 5 \times 10^{-9}$. 
1 Introduction

Much research has been done in recent years to find a more sensitive probe of cosmic strings in Cosmic Microwave Background (CMB) [1–5] and 21 cm intensity maps [6–10]. In our recent work [11] we presented a Bayesian interpretation of cosmic string detection and proposed a general machine learning framework to compute probability estimates for string locations on a CMB map and the posterior probability distribution for the string tension. In a machine learning context this is a supervised classification learning problem. Machine learning roughly divides into three branches, each concerning itself with a particular class of problems: supervised learning, which deals with learning from labeled data, unsupervised learning, which deals with extracting useful knowledge from unlabeled data and finally reinforcement learning, which deals with the general problem of learning goal-directed behaviour. (For a general introductions to machine learning with emphasis on supervised and unsupervised learning, see the textbook by Murphy [12] and the one by Hastie et al. [13], and for an introduction to reinforcement learning, see the textbook by Sutton and Barto [14].) The branch which concerns our work is supervised learning. Broadly speaking, we need to build a machine which classifies each pixel in the sky as being on a string or not being on a string. In our case this machine is a convolutional neural network and the building process is called training.

We implemented this approach with a convolutional neural network trained on simulations of CMB temperature anisotropy maps with and without strings and used it to estimated string locations on a CMB sky temperature map. (For an introduction to neural networks, see the recent book by Goodfellow et. al. [15].) We also showed a connection between these estimated string locations and the posterior probability of the string tension $G\mu$, given the sky map $\delta_{sky}$. In this paper we present the details of the convolutional neural network used in reference [11]. We begin in sections 2 and 3 with an introduction to supervised learning and neural networks, respectively. In section 4 we explain the details of the convolutional neural network we developed and in 5 we present the methodology used to train our network. In 6 we present our conclusions and the prospects for future work.
2 Supervised Learning

The supervised learning problem seeks to assign labels to data points given a dataset of example-label pairs. Suppose we have a dataset $D = \{(x_i, y_i) \mid i = 1\ldots N\}$, where $x_i$ and $y_i$ are the $i^{th}$ example and its associated label, belonging to some input space $A$ and some label space $B$, respectively. The goal of supervised learning is to find a mapping $f : A \to B$ which will generate the same labels as the process which originally generated $D$. The space $A$ is frequently $\mathbb{R}^n$, but not always. For example, $A$ could be the space of all graphs with labeled vertices. Note that we wish to approximate the unknown process which generates the dataset’s labels, not fit the dataset itself. A mapping which produces the right labels on the dataset but does not produce the right labels on another dataset produced by the same process is said to be overfitted. The standard way to detect overfitting is to split the dataset into a training set $D_{\text{train}}$ which we use to find the mapping $f$ and another evaluation set $D_{\text{eval}}$ on which we evaluate the performance of the mapping.

Let $f_w : A \to B$ be a manifold in function space parametrised by the coordinates $w$, the best choice of $w$ is the one which minimizes a problem-dependent error function, in general this function has the following form.

$$E(w) = \sum_{(x_i, y_i) \in D_{\text{train}}} ||y_i, f_w(x_i)||$$  \hspace{1cm} (2.1)

For example, if we set the input space $A = B = \mathbb{R}$, and consider the manifold in function space of all linear functions, and set the norm in the last equation to be the squared euclidean distance, this produces the standard linear regression problem, which is easily minimized by setting its derivatives to 0 and solving the resulting system of linear equations for $f_w$.

The goal of finding the best (for some definition of best) mapping among all functions which map from $A$ to $B$ is of course infeasible since the space is so large. The problem is made tractable by restricting ourselves to a finite dimensional manifold on this function space and finding the best function on the manifold, this can be interpreted as applying an infinitely strong prior on the function space. The whole supervised learning field concerns itself with finding the right priors in function space for a given problem. For instance if $A = B = \mathbb{R}^n$, the space of all linear functions from $A$ to $B$ is such a manifold, the prior here is simply zero for all nonlinear functions. Broadly speaking, supervised learning is about designing good functional forms and good error functions for a broad class of problems. In the next section we consider a class of functional forms called neural networks and discuss their properties in some detail.

Casting the problem of predicting string locations on an $m \times m$ pixel CMB map as a supervised learning problem, we make the following assignments: $A = \mathbb{R}^{m \times m}$, $B = \{0, 1\}^{m \times m}$. The dataset $D_{\text{train}} = \{(\delta_{\text{sky}}^i, \xi^i) \mid i = 1\ldots N\}$ is made up of $N$ simulated sky temperature maps, $\delta_{\text{sky}}^i$, at different $G\mu$ and the associated string location maps $\xi^i$. Given a pixel $j \equiv (j_1, j_2)$, $\xi_{j_1, j_2} = 1$ or 0, depending on whether a string is located at that pixel or not. In this context $f_w = \prod_{j \in \text{pixels}} (f_{w,j})^{\xi_j}(1 - f_{w,j})^{1-\xi_j}$ is a convolutional neural network, to be described in section 4, with free parameters labelled by $w$. And finally the error function used is the cross-entropy,

$$E(w) = \sum_{(\delta_{\text{sky}}^i, \xi^i) \in D_{\text{train}}} \sum_{j \in \text{pixels}} - \left\{ \xi_j \times \log(f_{w,j}(\delta_{\text{sky}}^i)) + (1 - \xi_j) \times \log(1 - f_{w,j}(\delta_{\text{sky}}^i)) \right\}.$$

\hspace{1cm} (2.2)
3 Neural Networks

In general, a neural network is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$ parametrized by some parameter vector $w$. These functions are often made up of alternating linear transformations (or rather affine functions) and element-wise nonlinearities. To obtain the output of the neural network, we first multiply the input $x \in \mathbb{R}^n$ by some matrix $W^1 \in \mathbb{R}^{h_1 \times n}$ whose elements are part of the parameter vector, then we apply an element-wise nonlinear function to the resulting vector $W^1x \in \mathbb{R}^{h_1}$, and repeat this process some number of times. This process of alternating linear transformations and element-wise nonlinearities produces "layers" in the functional form. In cases where there are many layers (here many can refer to as little as 3 and as many as 5000) such many-layered functions are called "deep". The ensemble of techniques related to such deep layers is called "Deep Learning". For a recent review of the field, see [16].

A simple, 1-layered example of such a form with $h_1$ "hidden units" which maps from $\mathbb{R}^n \rightarrow \mathbb{R}^m$ is

$$ F(x) = W^2\sigma(W^1x + b^1) + b^2, \quad (3.1) $$

where $W^1 \in \mathbb{R}^{h_1 \times n}$, $W^2 \in \mathbb{R}^{m \times h_1}$, $b^1 \in \mathbb{R}^{h_1}$ and $b^2 \in \mathbb{R}^m$, $\sigma(x)$ is some non-linear function which is applied element-wise to each element of its vector input. Here the elements of the parameter vector $w$ are all the elements of $W^1$ and $b^1$. If $\sigma$ is a non-constant, bounded, monotonic continuous function such a $\tanh(x)$ or the sigmoid($x$) $\equiv 1/(1 + \exp(-x))$, then functions of this class are universal approximators on the unit Hypercube [17] : for any piecewise continuous function $G(x)$ and some $\epsilon > 0$, there exists an $h_1$ and some parameters $w$ such that $F(x)$ approximates $G(x)$ to a better accuracy than $\epsilon$ (in either the mean-square or supremum norms). We can generalize the above functional form by allowing for multiple layers (a Multi-Layered Perceptron, or MLP for short). Let $F(x; w)$ be the form above parametrized by the vector of weights $w$, then we can define an n-layered neural network as such:

$$ F_{mlp}(x) = F_n(F_{n-1}(...F_1(x; w_1)...; w_{n-1}); w_n) \quad (3.2) $$

A simple 2-layer case of the form above gives: $F(x) = W^3\sigma(W^2\sigma(W^1x + b^1) + b^2) + b^3$.

Training such functions is the process of parameter fitting to some data using some error characteristic. Let $x_i$ be a set of $N$ "inputs" and $y_i$ be a set of $N$ "answers", then we define the error function as such $E(W) = \sum_i ||y_i - F_{mlp}(x_i; W)||$. If we use the least-square norm, the error function is quadratic in the parameters $W$ and we can exactly optimize by setting its partial derivatives to 0 and solving the subsequent system of linear equations. On the other hand, for a multi-layered function $F$, the error landscape is in general non-convex and has multiple local minima. Though non-trivial, it is nonetheless possible to optimize it with various heuristic methods [18], most of them using the concept of gradient descent.

A gradient evaluation of this function, for general non-linear functions of multiple variables, has the same computational cost as $N$ evaluations of the function, where $N$ is the dimension of the domain. However, for neural networks there exists an algorithm which allows for gradient computation to be roughly as costly as a single evaluation of the function. The algorithm is called "Backpropagation" [19] and is easily derived by applying the chain rule to equation 3.2 and rerouping terms. This decrease in the computational cost of evaluating first derivatives is one of the main advantages of neural networks over other function approximators.
4 The Convolutional Neural Network’s Parametrization

In [11] we used a convolutional neural network to produce prediction maps for string locations. These prediction maps were approximations to the pixel dependent probability \( p_j \) of there being a string at a pixel \( j \equiv (j_1, j_2) \) of a sky map, for a given sky map \( \delta_{sky} \) and string tension \( G\mu \). The prediction map produced by the network depended on series of parameters that we named \( \beta \) in [11] but which we call \( w \) in this paper’s notation. The values of \( w \) were chosen through the training of the network so that they gave the best approximation to \( p_j \) as defined through the Kullback-Leibler divergence given in equation 4.7 of reference [11]. Equation 4.7 from [11] is the cross-entropy we quoted in eq. 2.2 with \( f_{w, \beta}(\delta_{sky}) \) from eq. 2.2 corresponding to \( p_{\beta, j}(\delta_{sky}) \) in eq. 4.7 of [11]. In this section we expand on the nature of this convolutional neural network and explain the choices we made when we designed it.

The idea behind convolutional neural networks is to exploit the 2 dimensional structure of some data to drastically reduce the number of parameters we need to optimize. Consider using the function 3.1 for our task, here the input \( x \) are the pixel values of the sky temperature map, the \( W^1 \) matrix will have \( m^2 \times h_1 \) parameters, where \( m \) is the pixel side length of our map simulations. The functional form in 3.1 does not exploit the fact that the input is an image: the network will perform identically if we randomly permute the pixels in the map. To fix this, we impose an infinitely strong prior on the weight matrix \( W^1 \) and set most of its entries to 0. To achieve this, set \( h_1 = m^2 \). \( W^1 \) now has as many rows as we have pixels in the CMB maps, to each row \( i \) we assign a different pixel in the map, each column is also already associated with a pixel \( j \), we impose that the only nonzero entries of \( W^1 \) be those where the pixel \( j \) is in the neighborhood of pixel \( i \). Here the notion of neighbourhood is flexible, but for definiteness say that two pixels are neighbours if they are adjacent. We impose periodic boundary conditions on our 2-d maps and each pixel then has 8 adjacent pixels. With this definition each row of \( W^1 \) will only have 9 non-zero elements. We also impose that the weight matrix be translation invariant in the sense that \( W^1_{ij} = W^1_{i+k,j+k} \) for every translation by \( k \). Keep in mind that \( i, j, k \) here are pixels that are represented by pairs of integers. After these conditions are applied, there remain only 9 independant parameters in \( W^1 \), and \( W^1_{ij} = 0 \) for pixels \( j \) outside of the \( 3 \times 3 \) rectangle around \( i \). We see that \( W^1 \) is a convolution, \( W^1[i-j] \equiv W^1_{i,j} \) on the sky map, or a filter with kernel size 3 and stride size 1:

\[
(W^1 \star \delta_{sky})[i] = \sum_{i'=1}^{m^2} W^1[i-i'] \delta_{sky}[i'] \equiv \sum_{i'=1}^{m^2} W^1_{i,i'} \delta_{sky}[i'] \tag{4.1}
\]

We can also have a convolution operation \( V \) between vector-valued maps with vector dimensions \( A \) to \( B \). If each pixel \( i \) of the map \( x \) has a vector value of dimension \( A \) given by \( x[i]^\alpha \), with \( \alpha = 1, \ldots, A \), then we have the vector-valued map of vector dimension \( B \) given by:

\[
(V_\beta \star x)[i] = \sum_{\alpha=1}^{A} \sum_{i'=1}^{m^2} V_\beta[\alpha] x[i']^\alpha \tag{4.2}
\]

We now define 4 convolution operations \( W^\alpha \), \( a = 1, 2, 3, 4 \), between maps with pixels \( m \times m = 512 \times 512 \), and with the pixel values having vector-dimensions \( A_1, B_2 \). We take \( A_1 = 1 \) and all the other \( A \)’s and \( B \)’s are 32. We also pick a bias vector \( b_\beta \) of dimension 32 which we add to every pixel after each convolution. The values of \( W^\alpha_{\alpha, \beta} \) and \( b_\beta \) are free parameters that will be chosen when training the network.
The map \((W^1 * \delta_{\text{sky}}) + b^1\) will be an \(m \times m\) map with vector-dimension \(B_1\). On this map we apply \(\tanh\) element-wise so that \(\tanh(W^1 * \delta_{\text{sky}} + b^1)\) is still an \(m \times m\) map with vector-dimension \(B_1\). We repeat this process with all 4 convolutions:

\[
(tanh^4(W + b) * \delta_{\text{sky}}) \equiv \tanh(W^4 * \tanh(W^3 * \tanh(W^2 * \tanh(W^1 * \delta_{\text{sky}} + b^1) + b^2) + b^3) + b^4)
\] (4.3)

Finally we take the dot product of the vector value at each pixel with the vector \(\vec{c}\) to produce a scalar value at each pixel and again add a scalar bias \(b^5\) to each pixel. We then apply the \(\text{sigmoid}(x) = 1/(1 + \exp(-x))\) element-wise on this scalar valued map:

\[
f_{w,j}(\delta_{\text{sky}}) \equiv \text{sigmoid}\left(\vec{c} \cdot (tanh^4(W + b) * \delta_{\text{sky}}) + b^5\right)[j].
\] (4.4)

Here \(w\) represents the free parameters of this function. They are the elements of \(W^1, W^2, W^3, W^4, b^1, b^2, b^3, b^4, \vec{c}, \) and \(b^5\). These are the parameters we are modifying to optimise the cross-entropy defined in eq. 2.2. The function also has the "hyperparameters" \(A_a, B_a, \) and the \(3 \times 3\) kernel size which we have fixed in the presentation above. During regression those numbers are kept constant, but we used a small grid search to find which set of hyperparameters would give the best results. It is possible to understand the form above as the application of \(A_a\) different linear filters on the map \(x\), obtaining \(B_a\) new maps \(x'\). We then combined them using the local nonlinear function \(\tanh\) and \(\text{sigmoid}\).

An example of a common operation on maps which has this general form is the computation of the norm of the gradient: to compute it, we first apply the linear filters corresponding to taking derivatives in \(x\) and \(y\), we then combine those 2 derivative maps using a nonlinear function \(g(x, y) = \sqrt{(x)^2 + (y)^2}\).

We choose this particular functional form for our neural network for several of reasons. First, by the universal approximation theorem, this form is general enough to correctly approximate any nonlinear filter with support of size \(3 \times 3\). This generality implies that if there exists a nonlinear filter capable of determining whether a given pixel belongs to a cosmic string by looking at the surrounding \(3 \times 3\) pixels, then there exists some values of \(A\) and \(B\) for which 4.4 will be capable of approximating that filter arbitrarily closely. Furthermore, noticing that \(\tanh(x)\) is approximately linear for small values of \(x\), we can see that this form can also approximate linear filters by setting the \(W^a, b^a\) parameters to small values and the elements of \(\vec{c}\) to high values. We use the \(\text{sigmoid}(x)\) function because we wish to interpret the output of \(f_{w,j}(\delta)\) as the probability that a given pixel belongs to a string, probabilities must have values in \((0, 1)\).

Lastly, in image recognition applications, it is common to see functions with multiple convolutional layers (see [20] for an example of such a structure). Such functional forms contain nested convolutions and element-wise non-linearities, as we do. They also often contain so-called “pooling layers” which make the output invariant under small translations of the input map. We do not use these “pooling layers” because we found that they do not work well for our task. String detection from CMB maps requires the extraction of a very small signal from the overwhelming gaussian fluctuations and the smallness of this signal means that functions whose output is invariant under small translations of the input are not good detectors of strings.
5 Training and Implementation Details

As mentioned in section 2, the aim of training is to set the free parameters $w$ of the network discussed in the last section such as to minimize the cross-entropy eq. 2.2.

This minimization is achieved by a gradient descent algorithm which iteratively improves the free parameters in the direction of the negative gradient $\partial E/\partial w_j$. The parameters $w$ are initially drawn from a distribution with small mean and standard deviation to ensure that the training is better behaved. Starting with small values guarantees that the network is approximately linear at the beginning, so the derivatives will not start out too large or too small.

Computing $E(w)$ and $\partial E/\partial w_j$ on all maps in $D_{train}$ at every iteration is prohibitively expensive, since as we will discuss shortly, $D_{train}$ contains a rather large number of maps. This is solved by not using all of the training set for every parameter update iteration, but simply sampling some small number of maps and computing the gradient on those. This produces at every iteration a noisy estimates of the true gradient, the more maps we sample at each iteration, the closer this estimate is to the truth. Hence the training proceeds as follows:

1. Initiate $w$ randomly from a Gaussian distribution $\mathcal{N}(0, 0.01)$ and set the learning rate or step size $\alpha = 0.001$.
2. Randomly sample 10 input-answer tuples $(\delta_{sky}^i, \xi_i)$ from $D_{train}$, and compute $f_{w,j}(\delta_{sky}^i)$ using eq. 4.4 for each of the 10 samples.
3. Compute $E(w)$ for this set of ten $f_{w,j}(\delta_{sky}^i)$ and $\xi_i$ with eq. 2.2, and then also compute $\partial E/\partial w_j$.
4. Update $w_j$ to $w_j - \alpha \times \partial E/\partial w_j$.
5. Repeat from step 2 until the error function 2.2 no longer decreases. This was achieved after 1000 iterations. We call $\overline{w}$ the value of the parameters $w$ so obtained, and hence the network’s output map is $f_{\overline{w},j}(\delta_{sky})$.

In practice it is found that progressively decreasing the $\alpha$ parameter in step 4 is critical for converging to a good solution. There are many competing algorithms for scheduling $\alpha$ updates, we use the Adam algorithm [21] provided in the optimization package of the PyTorch environment. This algorithm keeps track of lower order moments of the gradients and uses them to set a parameter-dependent $\alpha$. Adam has two internal free parameters $\beta_1$ and $\beta_2$ which we respectively set to 0.9 and 0.999. These values have been found to adequately converge for our network. Ideally we could have perform random search over these parameters to obtain better convergence properties, however this would require retraining the network fully a large number of times and is therefore computationally infeasible for us.

As expected for a probability distribution, the map $f_{\overline{w},j}(\delta_{sky})$, has values between 0 and 1 for all pixels. This is ensured by the network architecture. However we need to normalize each such value to ensure that when the network predicts a probability of $\phi$ of being on a string, a fraction $\phi$ of them are actually on a string. Thus for a fixed $G_\mu$ we do the following:

1. Sample 100 sky maps from $D_{train}$ all with the same $G_\mu$ and their associated $\xi$ maps.
2. For every map $\delta_{sky}^i$ generated in the last step to obtain a dataset of 100 output maps $f_{\overline{w},j}(\delta_{sky}^i)$.
3. Break up the interval 0 to 1 into 1000 bins of size $\Delta \phi = 0.001$. For each bin between 0 and 1 compute the fraction of pixels in the 100 output maps with values in $[\phi, \phi + \Delta \phi]$ which contain strings. Let $h_{G \mu}(\hat{\phi})$ be this fraction.

4. We rescale the values of the map $f_{w,j}(\delta_{\text{sky}})$ by applying $h_{G \mu}$ element-wise to it. In this way we obtain our final prediction map $p_{w,j}(\delta_{\text{sky}}) \equiv h_{G \mu}(f_{w,j}(\delta_{\text{sky}}))$.

We perform the above procedure for 200 values of $G \mu$ between $10^{-11}$ and $10^{-6}$ in equally spaced log intervals: $G \mu = 10^{-11 + n \times 5/200}$, $n = 0, \ldots, 200$.

To make sure we have not overfitted and to test the performance of our trained network, we evaluated the loss function 2.2 on sky maps produced from gaussian maps and string maps that were not part of the training set. In particular, for the training set, the average cross entropy at the end of training was 0.67 with a standard deviation of 0.02. The cross entropy for maps in the evaluation set was indistinguishable from the training set. We used the PyTorch environment (pytorch.org) for machine learning and optimization algorithms. Training the model on a Tesla K80 GPU took 12 hours in total.

Furthermore, using the evaluation set, we produced the map $f_{w,j}(\delta_{\text{sky}})$ given by eq. 4.4 and apply $h_{G \mu}$ on it element-wise to get our prediction of the string locations $p_{w,j}(\delta_{\text{sky}})$. Figure 1 shows one such sky map, along with the location of the strings, and the convolutional neural network’s prediction $p_{w,j}(\delta_{\text{sky}})$. Figure 1 reproduces figures 1 and 2 from [11].

Finally, let us explain exactly what we mean when we say we sample from $D_{\text{train}}$. Every map in the training set is produced using the following equation: $\delta_{\text{sky}} = \delta_{\text{gauss}} + G \mu \delta_{\text{string}}$. As described in [11], we numerically produce a dataset of 500 gaussian maps $\delta_{\text{gauss}}$ and a dataset of 500 string temperature maps $\delta_{\text{string}}$ along with their associated answer maps $\xi^i$. These all
have a size of 512 \times 512 pixels with a resolution of 1 arcminute per pixel. We therefore have 500\(^2\) unique combinations of gaussian-string maps. Furthermore, since a translated gaussian map is still a possible gaussian map and there are 511\(^2\) possible translations of the maps. We effectively have 511\(^2\) \times 500\(^2\) distinct combinations. There still remains some freedom in the choice of \(G_\mu\), every time we want to sample from \(D_{\text{train}}\) we will generate the \(G_\mu\) by sampling from a prior distribution \(P(G_\mu)\). Since Planck quotes a 95\% confidence limit on the Nambu-Goto string tension of \(G_\mu < 1.3 \times 10^{-7}\), we chose a distribution that decreases exponentially in the string tension with 95\% of its area below \(G_\mu = 9 \times 10^{-8}\):

\[
P(G_\mu) = \frac{\exp(-G_\mu / (3 \times 10^{-8}))}{3 \times 10^{-8}}.
\]

(5.1)

Thus the procedure to sample maps from \(D_{\text{train}}\) is to first randomly choose a gaussian map, a string map, and a string tension from \(\{\delta_{\text{gauss}}^i\}, \{\delta_{\text{string}}^i\}\) and 5.1, respectively. Then we apply a random translation to the gaussian map and finally set \(\delta_{\text{sky}} = \delta_{\text{gauss}} + G_\mu \delta_{\text{string}}\).

6 Conclusions

In [11] we presented a bayesian interpretation of cosmic string detection and proposed a general machine learning framework which we implemented with a convolutional neural network. The network as applied to noiseless simulations of CMB maps with arcminute resolution was able to locate strings and accurately determine the value of the string tension for sky maps having strings with string tension as low as \(G_\mu = 5 \times 10^{-9}\). Here we presented the details of that convolutional neural network and how it was trained on simulations of CMB temperature anisotropy maps with and without strings. We also explained in detail how we used it to obtain prediction maps for the estimated string locations.

The results from this network are very promising and while there are many directions for further research, we wish to mention a couple which are of particular interest to us. First of all, we presented our framework within the context of cosmic string detection in CMB temperature anisotropy maps. However it could equally be applied to the detection of cosmic string wakes in 21 cm intensity maps [6–10]. Applying this framework to 21 cm intensity maps will require modifying the network presented here.

Secondly, our network was derived within the context of a long string model [22]. We described and justified this choice in [11]. A next step would be to consider realistic string simulations with noise similar to those simulations discussed in [23] and used to place the Planck constraints on the string tension [24]. Work in this direction has already begun [25] and preliminary results are encouraging. In figures 2b and 2c we show a string location map from a realistic simulation [26] similar to those in [23], and a prediction map produced by the 4-layer neural network presented here when applied on a sky map containing those strings with string tension \(G_\mu = 5 \times 10^{-8}\), fig. 2a. The prediction map was made on a sky map without noise. These maps are analogous to those presented in fig. 1.

Once noise is included, a more sophisticated network will be needed to detect the strings. We believe there is great possibility for the improvement of the network presented here, both by using better architecture design such as [27] and many more layers. Our network used 4 layers and 28,097 parameters, and to train the model it took 12 hours on one Tesla K80 GPU. In the modern deep learning context our network is relatively simple. The type of layers and number of layers and parameters used by our network resembles that used for character recognition twenty years ago [28]. The general trend in deep learning research has been to
Figure 2: Figure 2a shows a CMB anisotropy temperature map with cosmic strings from a realistic simulation [26] having a string tension $G\mu = 5 \times 10^{-8}$. In 2b we show the actual placement of such strings in the sky map. In 2c we show $p_{w,j}(\delta_{sky})$, our neural network’s prediction when analyzing 2a.

improve network performance with deeper networks. Modern convolutional networks, such as GoogLeNet [29], which are used for object classification, image segmentation, and evaluating Go board positions, have on the order of $10^2$ layers and $10^7$ free parameters. Training such networks requires multiple GPUs to be run for days or even weeks. Thus the main limiting factor in our network design will be the access to such computer resources.

Acknowledgments

We thank Christophe Ringeval for useful discussions and we thank François Bouchet and Christophe Ringeval for providing the CMB map and the string location map used in figures 2a and 2b. We would like to acknowledge the support of the Fonds de recherche du Québec – Nature et technologies (FRQNT) Programme de recherche pour les enseignants de collège. Computations were made on the supercomputer Helios from Université Laval, managed by Calcul Québec and Compute Canada. The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), the ministère de l’Économie, de la science et de l’innovation du Québec (MESI) and the Fonds de recherche du Québec – Nature et technologies (FRQNT).
References

[1] S. Amsel, J. Berger and R. H. Brandenberger, “Detecting Cosmic Strings in the CMB with the Canny Algorithm,” JCAP **0804**, 015 (2008) doi:10.1088/1475-7516/2008/04/015 [arXiv:0709.0982 [astro-ph]].

[2] A. Stewart and R. Brandenberger, “Edge Detection, Cosmic Strings and the South Pole Telescope,” JCAP **0902**, 009 (2009) doi:10.1088/1475-7516/2009/02/009 [arXiv:0809.0865 [astro-ph]].

[3] R. J. Danos and R. H. Brandenberger, “Canny Algorithm, Cosmic Strings and the Cosmic Microwave Background,” Int. J. Mod. Phys. D **19**, 183 (2010) doi:10.1142/S0218271810016324 [arXiv:0811.2004 [astro-ph]].

[4] L. Hertz, A. Amara, R. Brandenberger, T. Kacprzak and A. Refregier, “Searching for Cosmic Strings in CMB Anisotropy Maps using Wavelets and Curvelets,” [arXiv:1608.00004 [astro-ph]].

[5] J. D. McEwen, S. M. Feeney, H. V. Peiris, Y. Wiaux, C. Ringeval and F. R. Bouchet, “Wavelet-Bayesian inference of cosmic strings embedded in the cosmic microwave background,” [arXiv:1611.10347 [astro-ph.IM]].

[6] R. H. Brandenberger, R. J. Danos, O. F. Hernández and G. P. Holder, “The 21 cm Signature of Cosmic String Wakes,” JCAP **1012**, 028 (2010) [arXiv:1006.2514 [astro-ph.CO]].

[7] O. F. Hernández, Y. Wang, R. Brandenberger and J. Fong, “Angular 21 cm Power Spectrum of a Scaling Distribution of Cosmic String Wakes,” JCAP **1108**, 014 (2011) [arXiv:1104.3337 [astro-ph.CO]].

[8] O. F. Hernández and R. H. Brandenberger, “The 21 cm Signature of Shock Heated and Diffuse Cosmic String Wakes,” JCAP **1207**, 032 (2012) [arXiv:1203.2307 [astro-ph.CO]].

[9] O. F. Hernández, “Wouthuysen-Field absorption trough in cosmic string wakes,” Phys. Rev. D **90**, no. 12, 123504 (2014) doi:10.1103/PhysRevD.90.123504 [arXiv:1403.7522 [astro-ph.CO]].

[10] R. H. Brandenberger, O. F. Hernández and D. C. N. da Cunha, “Disruption of Cosmic String Wakes by Gaussian Fluctuations,” Phys. Rev. D **93**, no. 12, 123501 (2016) doi:10.1103/PhysRevD.93.123501 [arXiv:1508.02317 [astro-ph.CO]].

[11] R. Ciuca and O. F. Hernández, “A Bayesian Framework for Cosmic String Searches in CMB Maps,” JCAP **1708**, no. 08, 028 (2017) doi:10.1088/1475-7516/2017/08/028 [arXiv:1706.04131 [astro-ph.CO]].

[12] K.P. Murphy, “Machine learning: a probabilistic perspective,” 2012, MIT Press, Cambridge, MA.

[13] T. Hastie, R. Tibshirani, J. Friedman, “The Elements of Statistical Learning,” 2009, Springer, New York.

[14] R. Sutton, A. Barto, “Introduction to Reinforcement Learning,” 1998, MIT Press, Cambridge, MA.

[15] I. Goodfellow, Y. Bengio, A. Courville, “Deep Learning” 2017, MIT Press, Cambridge, MA.

[16] Y. LeCun, Y. Bengio, and G. Hinton. “Deep learning”. Nature, 521:436-444, May 2015.

[17] K. Hornik, M. Stinchcombe, H. White, “Multilayer feedforward networks are universal approximators” Neural Networks, 1989, 2, (5), pp. 359-366.

[18] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B. & LeCun, Y. “The loss surface of multilayer networks”. In Proc. Conference on AI and Statistics, [arXiv:1412.0233].

[19] Rumelhart, D. E., Hinton, G. E., Williams, R. J. “Learning representations by back-propagating errors”. Nature 323, 533-536 (1986).
[20] Turaga, S. C. et al. “Convolutional networks can learn to generate affinity graphs for image segmentation”. Neural Comput. 22, 511-538 (2010).

[21] D.P. Kingma, J. Ba, “Adam: A Method for Stochastic Optimization” (2014), [arXiv:1412.6980 [cs.LG]]

[22] L. Perivolaropoulos, “COBE versus cosmic strings: An Analytical model,” Phys. Lett. B 298, 305 (1993), [hep-ph/9208247].

[23] A. A. Fraisse, C. Ringeval, D. N. Spergel and F. R. Bouchet, “Small-Angle CMB Temperature Anisotropies Induced by Cosmic Strings,” Phys. Rev. D 78, 043535 (2008) doi:10.1103/PhysRevD.78.043535 [arXiv:0708.1162 [astro-ph]].

[24] P. A. R. Ade et al. [Planck Collaboration], “Planck 2013 results. XXV. Searches for cosmic strings and other topological defects,” Astron. Astrophys. 571, A25 (2014) doi:10.1051/0004-6361/201321621 [arXiv:1303.5085 [astro-ph.CO]].

[25] O. F. Hernández, C. Ringeval, et al. work in progress.

[26] C. Ringeval, private communication.

[27] K. He, X. Zhang, S. Ren, J. Sun, “Deep Residual Learning for Image Recognition”. CVPR 2016: 770-778, [arXiv:1512.03385].

[28] Y. LeCun, L. D. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. A. Muller, E. Sackinger, P. Simard and V. Vapnik: “Learning Algorithms For Classification: A Comparison On Handwritten Digit Recognition”, in Oh, J. H. and Kwon, C. and Cho, S. (Eds), Neural Networks: The Statistical Mechanics Perspective, 261-276, World Scientific, 1995.

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich., “Going Deeper with Convolutions”, 2015, CVPR, page 1-9. IEEE Computer Society, [arXiv:1409.4842].