Deep Sequential Neural Networks

Abstract

Neural Networks sequentially build high-level features through their successive layers. We propose a new neural network model where each layer is associated with a set of candidate mappings. When an input is processed, at each layer, one mapping among these candidates is selected according to a sequential decision process. The resulting model is structured according to a DAG like architecture, so that a path from the root to a leaf node defines a sequence of transformations. The model is thus able to process data with different characteristics through specific sequences of local transformations, increasing the expression power of this model w.r.t a classical deep neural network. The learning algorithm is inspired from policy gradient techniques coming from the reinforcement learning domain and is used here instead of the classical back-propagation based gradient descent techniques.

Keywords: reinforcement learning, deep learning, policy gradient

1. Introduction

Reinforcement Learning (RL) techniques which are usually devoted to problems in dynamic environments have been recently used for classical machine learning tasks like classification (Dulac-Arnold et al., 2014; Busa-Fekete et al., 2012). In that case, the prediction process is seen as a sequential process, and this sequential process can take different forms. For example Dulac-Arnold et al. (2011) and Mnih et al. (2014) consider that the sequential process is an acquisition process able to focus on relevant parts of the input data. RL opens now some interesting research directions for classical ML tasks and allows one to imagine solutions to complex problems like budgeted classification (Dulac-Arnold et al., 2012) or anytime prediction (Farhangfar et al., 2009).

In parallel, Neural Networks (NNs) have recently given rise to a large amount of research motivated by the development of deep architectures - or Deep Neural Networks (DNNs). The use of deep architectures have shown impressive results for many different tasks, from image classification (Krizhevsky et al., 2012), speech recognition (Graves et al., 2013) or even for machine translation (Zou et al., 2013). These great successes mainly come from the ability of DNNs to compute high-level features over data. Many variants of learning algorithms have been proposed but the baseline learning algorithm still consists in recursively computing the gradient by using the back-propagation algorithm and performing (stochastic) gradient descent.

This paper is motivated by the idea of using sequential learning algorithms - mainly coming from the reinforcement learning community - in the context of Deep Neural Networks. More precisely, we consider that inference in a NN is a sequential decision process which selects at each layer of a deep architecture one mapping among a set of candidate mappings. This process is repeated layerwise until the final layer is reached. The resulting NN is then a DAG like architecture, where each layer is composed of a set of candidate mappings. Only one of these candidates will be selected at each layer, for processing an input pattern. When an input is presented to the NN, it will then follow a set of successive transformations which corresponds to a trajectory in the NN DAG, until the final output

is computed. The decision on which trajectory to follow is computed at each layer through additional components called here selection functions. The latter are trained using a policy gradient technique like algorithm while the NN weights are trained using back propagation. This model called *Deep Sequential Neural Networks* (DNNs) process an input through successive local transformations insetad of using a global transformation in a classical deep NN architecture. It can be considered as an extension of the classical deep NN architecture since when the number of potential candidate mapping at each layer is reduced to 1, one recovers a classical NN architecture. DSNNs are thus based on the following inference process: (i) Given an input x, the model chooses between different possible mappings[1] (ii) Then x is mapped to a new representation space.(iii) Given the new representation, another mapping is chosen between a set of different possible mappings, and so on to the prediction.

Note that the way mappings are chosen, and the mappings themselves are learned together on a training set. Instead of just computing representations in successive representation spaces, DSNNs are able to choose the best representation spaces depending on the input and to process differently data coming from different distributions.

This idea of choosing a different sequence of computations depending on the input share many common points with existing models (see Section 5) but our model has some interesting properties:

- It is able to simultaneously learn successive representations of an input, and also which representations spaces are the most relevant for this particular input.

- Learning is made by extending **policy gradient** methods which as far as we know have never been used in this context; moreover, we show that, when the DNNs is in its simplest shape, this algorithm is equivalent to the gradient descent technique used in NNs.

The paper is organized as follows: in Section 2, we describe the DSNN formalisms and the underlying sequential inference process. By deriving a policy gradient algorithm, we propose a learning algorithm in Section 3 based on gradient descent techniques. We present in Section 4 experimental results on different datasets and a qualitative study showing the ability of the model to solve complex classification problems. The related work is presented in Section 5.

2. Deep Sequential Neural Networks

Let us consider $\mathcal{X} = \mathbb{R}^X$ the input space, and $\mathcal{Y} = \mathbb{R}^Y$ the output space, X and Y being respectively the dimension of the input and output spaces. We denote $\{(x_1, y_1), ..., (x_\ell, y_\ell)\}$ the set of labeled training instances such that $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. $\{(x_{\ell+1}, y_{\ell+1}), ..., (x_T, y_T)\}$ will denote the set of testing examples. The DSNN model has a DAG-structure defined as follow:

- Each node n is in $\{n_1, ..., n_N\}$ where N is the total number of nodes of the DAG. The root node is n_1, n_1 does not have any parent node. $c_{n,i}$ corresponds to the i-th

1. We call a mapping the base transformation made between two layers of a neural network i.e a projection from $\mathbb{R}^n$ to $\mathbb{R}^m$.

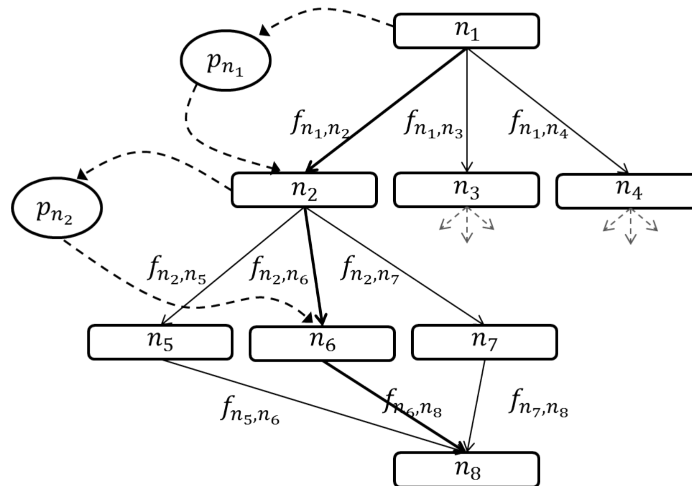

Figure 1: Architecture of *Deep Sequential Neural Networks*. We illustrate a model where each node has 3 children. For a particular input, and by using p_{n_1} and p_{n_2}, the sequence of chosen nodes is (n_1, n_2, n_6, n_8). Note that only p_{n_1} and p_{n_2} have been illustrated on the figure but each node is associated with a p function. In that case, the final prediction given x is $f_{n_6,n_8}(f_{n_2,n_6}(f_{n_1,n_2}(x)))$.

child of node n and $\#n$ is the number of children of n so, in that case, i is a value be between 1 and $\#n$. $leaf(n)$ is *true* if node n is a leaf of the DAG - i.e a node without children.

- Each node is associated to a particular representation space $\mathbb{R}^{dim(n)}$ where $dim(n)$ is the dimension associated to this space. Nodes play the same role than layers in classical neural networks. $dim(n_1) = X$ i.e the dimension of the root node is the dimension of the input of the model. For any node n, $dim(n) = Y$ if $leaf(n) = true$ i.e the dimension of the leaf nodes is the output space dimension.

- We consider *mapping functions* $f_{n,n'} : \mathbb{R}^{dim(n)} \rightarrow \mathbb{R}^{dim(n')}$ which are functions associated with edge (n, n'). $f_{n,n'}$ computes a new representation of the input x in node n' given the representation of x in node n. The output produced by the model is a sequence of f-transformation applied to the input like in a neural network.

- In addition, each node is also associated with a *selection function* denoted $p_n : \mathbb{R}^{dim(n)} \rightarrow \mathbb{R}^{\#n}$ able, given an input in $\mathbb{R}^{dim(n)}$, to compute a score for each child of node n. This function defines a probability distribution over the children nodes of n such as, given a vector $z \in \mathbb{R}^{dim(n)}$ using softmax transformation. Selection functions aim at selecting which f-functions to use by choosing a path in the DAG from the root node to a leaf node.

Algorithm 1 DSNN Inference Procedure

```
1: procedure INFERENCE(x)                                    ▷ x is the input vector
2:     z^(1) ← x
3:     n^(1) ← n_1
4:     t ← 1
5:     while not leaf(n^(t)) do                              ▷ Inference finished
6:         a^(t) ~ p_{n^(t)}(z^(t))     ▷ Sampling using the distribution over children nodes
7:         n^(t+1) ← c_{n^(t),a^(t)}
8:         z^(t+1) ← f_{n^(t),n^(t+1)}(z^(t))
9:         t ← t + 1
10:    end while
11:    return z^(t)
12: end procedure
```

2.1 Inference in DSNN

Given such a DAG structure $\mathcal{G}$, the inference process is the following: At first, an input $x \in \mathcal{X}$ is presented at the root node $n^{(1)} = n_1$ of the DAG[2]. Then, based on x, a child node $n^{(2)}$ is sampled using the $P(c_{(1)_{,.}}|x)$ distribution computed through the p_{n_1} function. The model computes a new representation at node $n^{(2)}$ using $f_{n^{(1)},n^{(2)}}(x)$. A child node of $n^{(2)}$ is sampled following $P(c_{(2)_{,.}}|x)$, ... The same process is repeated until a leaf node. The vector computed at the leaf node level is the output of the model.

Details of the inference procedure are given in Algorithm 1. The algorithm is a discrete-time sequential process starting at time $t = 1$ and finishing when the input has reached a leaf node. Given an input x, we denote:

- $n^{(t)}$ the node reached by the input x at time t such that $n^{(1)} = n_1$.
- $a^{(t)}$ the child node chosen at time t, $a^{(t)} \in [1..\#n^{(t)}]$
- $z^{(t)}$ the mapping of x at time t such that $z^{(t)} \in \mathbb{R}^{dim(n^{(t)})}$

The inference process generates a trajectory T which is a sequence $(n^{(1)}, ..., n^{(D)})$ of nodes starting from the root node $n^{(1)} = n_1$ to a leaf of the node $n^{(D)}$ such that $leaf(n^{(D)}) = True$; D is the size of the chosen branch of the tree. This sequence is obtained by sequentially choosing a sequence of children (or actions) $H = (a^{(1)}, ..., a^{(D-1)})$. In the following H will denote a sequence of actions sampled w.r.t the p functions.

3. Learning DSNN with gradient-based approaches

The training procedure we propose aims at simultaneously learning both the *mapping functions* $f_{i,j}$ and the *selection functions* p_i in order to minimize a given learning loss denoted Δ. Our learning algorithm is based on an extension of **policy gradient techniques** inspired from the Reinforcement Learning literature. More precisely, our learning method is close to the methods proposed in Wierstra et al. (2007) and Mnih et al. (2014) with the difference that, instead of considering a reward signal which is usual in reinforcement learning, we

2. $n^{(t)}$ is used to denote the node selected at time t

consider a loss function Δ computing the quality of the system.

Let us denote θ the parameters of the f functions and γ the parameters of the p functions. The performance of our system is denoted $J(\theta, \gamma)$:

$$J(\theta, \gamma) = E_{P(x,H,y)}[\Delta(F(x, H), y)] \tag{1}$$

where both H - i.e the sequence of children nodes chosen by the p-functions - and F - the final decision given a particular path in the DSNN - depends on both γ and θ. The optimization of J can be made by gradient-descent techniques and we need to compute the gradient of J:

$$\nabla_{\theta,\gamma} J(\theta, \gamma) = \int \nabla_{\theta,\gamma} \left(P(H|x)\Delta(F(x, H), y) \right) P(x, y) dH dx dy \tag{2}$$

This gradient can be simplified such that:

$$\begin{aligned} \nabla_{\theta,\gamma} J(\theta, \gamma) &= \int \nabla_{\theta,\gamma} \left(P(H|x) \right) \Delta(F(x, H), y) P(x, y) dH dx dy \\ &+ \int P(H|x) \nabla_{\theta,\gamma} \Delta(F(x, H), y) P(x, y) dH dx dy \\ &= \int P(H|x) \nabla_{\theta,\gamma} \left(log P(H|x) \right) \Delta(F(x, H), y) P(x, y) dH dx dy \\ &+ \int P(H|x) \nabla_{\theta,\gamma} \Delta(F(x, H), y) P(x, y) dH dx dy \end{aligned} \tag{3}$$

Using the Monte Carlo approximation of this expectation by taking M trail histories over the training examples, we can write:

$$\nabla_{\theta,\gamma} J(\theta, \gamma) = \frac{1}{\ell} \sum_{i=1}^{\ell} \left[\frac{1}{M} \sum_{k=1}^{M} \nabla_{\theta,\gamma} \left(log P(H|x_i) \right) \Delta(F(x_i, H), y) + \nabla_{\theta,\gamma} \Delta(F(x_i, H), y) \right] \tag{4}$$

Intuitively, the gradient is composed of two terms: (i) The first term aims at penalizing trajectories with high loss - and thus encouraging to find trajectories with low loss. When the loss is 0, the resulting gradient is null and the system will thus continue to choose the same paths. (ii) The second term is the gradient computed over the branch of the tree that has been sampled. It encourages the f functions to perform better the next time the same path will be chosen for a particular input. While the second term can be easily computed by back-propagation techniques over the sequence of f functions that compose the branch of the tree, the computation of $\nabla_{\theta,\gamma} log P(H|x_i)$ can be written:

$$\nabla_{\theta,\gamma} log P(H|x_i) = \nabla_{\theta,\gamma} \sum_{t=1}^{D} log P(a^{(t)}|z^{(t)}) \tag{5}$$

The term $\nabla_{\theta,\gamma} log P(a^{(t)}|z^{(t)})$ depends on $z^{(t)}$ which is the projection of the input x at node $n^{(t)}$. This projection involves the sequence of transformation $f_{n^{(1)},n^{(2)}}, ..., f_{n^{(t-1)},n^{(t)}}$ and the selection function $p_{n^{(t)}}$. It can also be computed by back-propagation techniques over the functions $f_{n^{(1)},n^{(2)}}, ..., f_{n^{(t-1)},n^{(t)}}, p_{n^{(t)}}$.

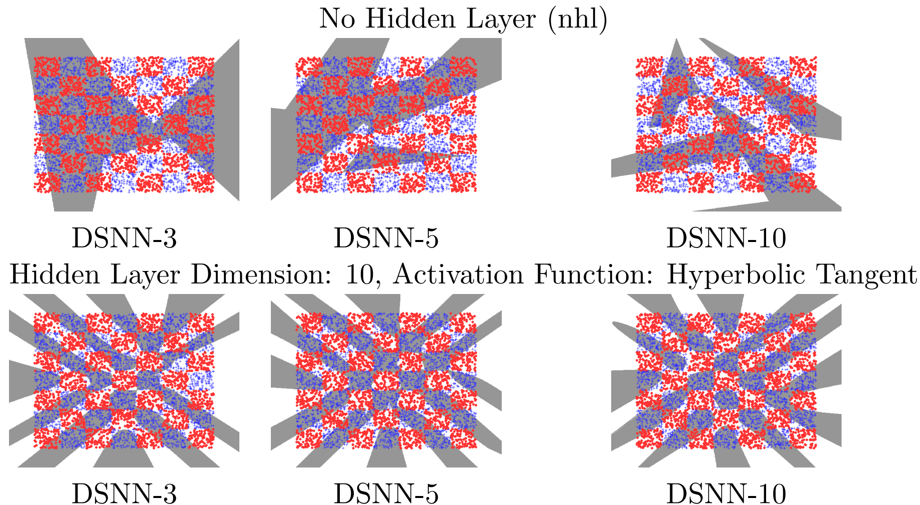

Table 1: Examples of Decision Frontiers obtained on the Checkboard 7×7 dataset

Variance reduction: Note that equation provides us an estimate of the gradient which can have a high variance. Instead of using this estimate, we replace $\Delta(F(x_i, H), y)$ by $\Delta(F(x_i, H), y) - b$ where $b = E_{p(x,H,y)}[\Delta(F(x_i, H), y)]$ which can be easily estimated on the training set (Wierstra et al., 2007).

4. Experiments

We have performed experiments comparing two different models: (i) **NN** corresponds to a simple neural network (ii) **DSNN-k** corresponds to the sequential model presented above where k is the number of possible actions. It corresponds to the extension of a NN with a 5-dimensionnal hidden layer (with hyperbolic tangent activation function) where now the system is able to choose at each timestep between 2 actions. *nhl* will denote a model without hidden layer. *DSNN-3 10-10 (rl)* corresponds to the extension of a NN with two hidden layers of size 10 (with rectified linear units) with 3 possible actions. The f functions are thus linear transformations followed by a non linear function. The $p_{.}$ functions are simple linear functions[3].

The experiments have been made on three families of datasets. The first set of experiments has been made on 5 UCI datasets which are datasets composed of about 1,000 training examples in low-dimensional space. The second set of experiments has been made on a variation of MNIST where the distribution of the inputs has been pertubated to measure the ability of the system to computes different features depending on the inputs. At last, the third set of experiments on simple 2-dimensionnal datasets based on checkerboard distributions and is used to better analyze the behavior of the model. The experiments have been performed with many different values of the hyper-parameters following a grid search algorithm. For each value of hyper-parameters and each dataset, we have performed 5 runs, we have averaged the performance over the 5 runs.

3. More complex p functions could be used but have not been investigated in the paper

	Hidden Layer(s)	diabetes	fourclass	heart	sonar	splice
nhl	NN	* **78.4**	67.1	* **81.1**	**67.3**	69.0
	DSNN-2	76.3	70.5	77.5	64.1	**72.7**
	DSNN-5	77.6	73.5	76.5	65.1	68.1
	DSNN-10	77.8	**74.7**	77.9	66.3	65.8
5	NN	75.8	76.5	69.6	**79.7**	60.0
	DSNN-2	75.2	**94.5**	**74.3**	77.1	68.7
	DSNN-5	**76.9**	92.1	72.8	79.4	70.5
	DSNN-10	76.5	92.7	70.3	77.5	**69.8**
10	NN	**74.4**	77.9	69.4	78.7	61.1
	DSNN-2	73.9	* **95.7**	66.9	78.4	71.2
	DSNN-5	73.8	93.3	67.4	* **81.0**	**72.7**
	DSNN-10	74.0	93.8	**70.8**	77.5	68.9
25	NN	**77.1**	77.0	68.1	**77.5**	61.3
	DSNN-2	73.3	**94.4**	72.8	77.1	67.8
	DSNN-5	72.6	93.1	**73.3**	75.9	69.1
	DSNN-10	73.4	91.0	70.6	76.8	**71.4**
5-5	NN	73.3	90.2	75.0	**75.2**	64.5
	DSNN-2	72.9	**95.3**	75.0	73.0	* **74.8**
	DSNN-5	72.0	84.6	**76.0**	74.3	72.0
	DSNN-10	**73.8**	60.8	69.6	62.5	63.7
10-10	NN	70.1	**92.8**	77.2	76.8	64.6
	DSNN-2	**73.6**	90.0	73.5	75.2	71.0
	DSNN-5	72.0	88.7	**77.7**	76.2	**73.5**
	DSNN-10	73.0	67.4	76.7	**79.4**	67.4
25-25	NN	71.9	**93.7**	75.0	74.3	63.8
	DSNN-2	71.3	81.3	73.5	**76.5**	**72.8**
	DSNN-5	68.7	85.2	**77.2**	73.0	70.1
	DSNN-10	**74.3**	72.8	57.6	73.0	69.3

Table 2: Accuracy over UCI datasets (with tanh activation function). $*$ is the best results obtained for each dataset. **Bold** values corresponds to the best performance obtained for each architecture. Results are average over 5 runs.

	NN	DSNN-2	DSNN-3	DSNN-5
nhl	89.4	89.4	89.4	89.3
25	93.7	93.6	94.2	93.9
25-25	93.6	93.4	93.5	93.4
100	95.3	95.4	95.3	95.4
100-100	94.6	94.6	94.7	94.4

	NN	DSNN-2	DSNN-3	DSNN-5
nhl	27.7	88.3	88.2	**88.4**
5	37.4	82.6	**83.5**	56.7
10	83.4	**89.2**	85.6	87.7
10-10	81.1	**85.3**	84.0	82.9
25	**91.9**	91.5	91.0	91.4
25-25	**90.9**	90.4	85.1	78.3
50-50	92.8	**93.5**	92.9	79.3

Table 3: Accuracy on the MNIST dataset (left) and the MNIST-Negative dataset (right) - digits have been resampled to 14×14 images. We have used rectified linear units on the hidden layers.

UCI datasets: The results obtained on UCI datasets are presented in Table 2 where 50 % of the examples have been used for training. First, one can see that, for some datasets (diabetes,heart), a simple linear model is sufficient for computing a high accuracy and using more complex architectures does not help in increasing the performance of the models. In that case, using DSNN does not seem really useful since a simple model is enough. For the other datasets, the DSNN outperforms the NN approach, particularly when the number of children for each node is low. Indeed, when this number becomes high, the number of parameters to learn can be very large, and the system is not able to learn these parameters, or needs at least much more iterations to converge.

MNIST datasets: We have performed experiments on both the classical MNIST dataset[4] where digits have been re-sampled to 14×14 images, and to a variation of this

4. The training set is composed of 12,000 examples, and the testing set is composed of 50,000 digits.

dataset called MNIST-Negative where half of the digits have been negated - i.e for half of the digits, the value of a pixel is equal to one minus its original value. In that case, one can consider that digits have been sampled following two different distributions a simple model will not be able to capture. Table 3 reports the results we have obtained with different architectures. First, one can see that, for the MNIST dataset, the performance of NN and DSNN are quite similar showing that DSNN is not relevant when the input distribution is simple. On the MNIST-Inverse dataset, first, the NN without hidden layer is unable to well classify since the inputs are too much disparate. In that case, DSNN is able to capture the two inputs distributions and performs quite well. Adding some small hidden layers allows us to increase the accuracy. When using large hidden layers, a single NN is capable of capturing the data distribution and thus perform as well as DSNN.

Checkerboard datasets: Figure 1 shows the decision frontiers obtained by different architectures over a 7×7 checkerboard. This figures illustrates that the DSNN architecture introduces string non-linearities in the decision function with shallow architectures, DSNN being an alternative to deep neural networks. Note that DSNN does not suffer of gradient vanishing problems since a depth of 1 or 2 is sufficient for over-fitting complex datasets.

5. Related Work

Different models are related to DSNNs. The first family of models are neural networks. The idea of processing input data by different functions is not new and have been proposed for example in Neural Tree Networks (Utgoff, 1988; Sirat and Nadal, 1990), with Hierarchical Mixture of Experts (Jordan and Jacobs, 1994) where the idea is to compute different transformations of data and to aggregate these transformations. The difference with our approach is both in the inference process, and in the way the model is learned. They also share the idea of processing different inputs with different computations which is the a major idea underlying decision trees (Quinlan, 1986) and also more recent classification techniques like Dulac-Arnold et al. (2011). At last, some links have already be done between classification and reinforcement learning algorithms (Dulac-Arnold et al., 2012; Busa-Fekete et al., 2012). Particularly, the use of recurrent neural networks from modelling Markov Decision Processes learned by Policy gradient techniques has been deeply explored in Wierstra et al. (2007) and in a recent work that proposes the use of such models for image classification (Mnih et al., 2014).

6. Conclusion and Perspectives

We have proposed a new family of model called *Deep Sequential Neural Networks* which differ from neural networks since, instead of always applying the same set of transformations, they are able to choose which transformation to apply depending on the input. The learning algorithm is based on the computation of the gradient which is obtained by an extension of policy-gradient techniques. In its simplest shape, DSNNs are equivalent to DNNs. Experiments on different datasets have shown the effectiveness of these models.

References

Róbert Busa-Fekete, Djalel Benbouzid, and Balázs Kégl. Fast classification using sparse decision dags. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*, 2012. URL `http://icml.cc/discuss/2012/501.html`.

Gabriel Dulac-Arnold, Ludovic Denoyer, and Patrick Gallinari. Text classification: A sequential reading approach. In *Advances in Information Retrieval - 33rd European Conference on IR Research, ECIR 2011, Dublin, Ireland, April 18-21, 2011. Proceedings*, pages 411–423, 2011.

Gabriel Dulac-Arnold, Ludovic Denoyer, Philippe Preux, and Patrick Gallinari. Sequential approaches for learning datum-wise sparse representations. *Machine Learning*, 89(1-2):87–122, 2012.

Gabriel Dulac-Arnold, Ludovic Denoyer, Nicolas Thome, Matthieu Cord, and Patrick Gallinari. Sequentially generated instance-dependent image representations for classification. *Internation Conference on Learning Representations - ICLR 2014*, 2014.

Alireza Farhangfar, Russell Greiner, and Csaba Szepesvári. Learning to segment from a few well-selected training images. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, page 39, 2009.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 6645–6649, 2013.

Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Comput.*, 6(2):181–214, March 1994.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114, 2012.

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. *CoRR*, abs/1406.6247, 2014. URL `http://arxiv.org/abs/1406.6247`.

J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

J A Sirat and J-P Nadal. Neural trees: a new tool for classification. *Network: Computation in Neural Systems*, 1(4):423–438, 1990. doi: 10.1088/0954-898X_1_4_003.

Paul E. Utgoff. Perceptron trees: A case study in hybrid concept representations. In *Proceedings of the 7th National Conference on Artificial Intelligence. St. Paul, MN, August 21-26, 1988.*, pages 601–606, 1988. URL `http://www.aaai.org/Library/AAAI/1988/aaai88-107.php`.

Daan Wierstra, Alexander Förster, Jan Peters, and Jürgen Schmidhuber. Solving deep memory pomdps with recurrent policy gradients. In *Artificial Neural Networks - ICANN 2007*, pages 697–706, 2007.

Will Y. Zou, Richard Socher, Daniel M. Cer, and Christopher D. Manning. Bilingual word embeddings for phrase-based machine translation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 1393–1398, 2013.