Université de Montréal

Towards Deep Semi Supervised Learning

par Mohammad Pezeshki

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2016

Résumé

L'apprentissage profond est une sous-discipline de l'intelligence artificielle en plein essor grâce à d'impressionnantes performances, obtenue durant la dernière décennie, dans divers domaines d'application de l'apprentissage machine. Le pré-entraînement non supervisé des réseaux de neurones constitue une composante essentielle de ce succès. L'investigation d'idées combinant l'apprentissage supervisé et non supervisé se présente donc comme une étape naturelle.

Le réseau de neurones à échelles est une récente architecture semi-supervisée ajoutant une composante non supervisée à la perte supervisée des réseaux profonds. Le modèle peut être compris comme étant une partie d'une juxtaposition d'autoencodeurs debruitant apprenant à reconstruire chaque couche. Pour ce faire, la reconstruction est atteinte en considérant une corruption de la couche présente grâce aux retours des couches supérieures.

Le présent mémoire entreprend une analyse et déconstruction systématique de la performance des réseaux de neurones à échelles. Ainsi, nous analysons dix-neuf variantes de l'architecture obtenues en isolant les différentes composantes du modèles. Dans les chapitres I et II, nous introduisons les fondamentaux des réseaux de neurones, leur entraînement par descente de gradient, et leurs applications à l'apprentissage des représentations.

Dans les chapitres III et IV, nous offrons une comparaison exhaustive d'un grand nombre de variantes du réseau de neurones à échelles en contrôlant les hyper paramètres ainsi que la sélection d'ensemble de données. Au cours de notre investigation, nous découvrons certaines propriétés générales du modèle qui le distingue des habituels réseaux à propagation avant. Nous terminons par l'introduction d'une variante du réseau à échelles obtenant ainsi des résultats dépassant l'état de l'art actuel dans des tâches de classification supervisé et semi-supervisé sur la version invariante aux permutations de MNIST.

Mots clés: réseaux de neurones, apprentissage automatique, apprentissage de représentations profondes, apprentissage de représentations, apprentissage non supervisé, apprentissage supervisé, apprentissage semi-supervisé, régularisation

Summary

Deep Learning is a quickly growing area of research in the field of Artificial Intelligence that has achieved impressive results in the last decade in various Machine Learning applications. Unsupervised learning for pre-training layers of neural networks was an essential part of the first wave of deep learning methods. A natural next step is to investigate ideas that could combine both unsupervised and supervised learning.

The Ladder Network is a recently proposed semi-supervised architecture that adds an unsupervised component to the supervised learning objective of a deep network. The model can be seen as part of a deep stack of denoising autoencoders or DAEs that learns to reconstruct each layer. At each layer, the reconstruction is done based on a corrupted version of the current layer, using feedback from the upper layer.

This thesis undertakes a systematic analysis and deconstruction of the Ladder Network, investigating which components lead to its excellent performance. We analyze nineteen different variants of the architecture derived by isolating one component of the model at a time.

In Chapters 1 and 2, we introduce fundamentals of artificial neural networks, the gradient-based way of training them and their application in representation learning. We also introduce deep supervised and unsupervised learning and discuss the possible ways of combining them.

In Chapters 3 and 4, we provide a thorough comparison of a large number of variants of the Ladder Network controlling both hyperparameter settings and data set selection. Through our investigation, we discover some general properties of the model that distinguish it from standard feedforward networks.

Finally, we introduce a variant of the Ladder Network that yields to state-of-the-art results for the Permutation-Invariant MNIST classification task in both semi- and fully- supervised settings.

Keywords: neural networks, machine learning, deep learning, representation learning, unsupervised learning, supervised learning, semi-supervised learning, model regularization

Contents

List of Figures

List of Tables

List of Abbreviations

AE	Auto-Encoder
AMLP	Augmented Multi-Layer Perceptron
ANN	Artificial Neural Network
BN	Batch Normalization
CE	Cross Entropy
DAE	Denoising Auto-Encoder
dRBM	discriminant Restricted Boltzmann Machine
GD	Gradient Descent
I.I.D	Independent and Identically Distributed
INIT	Initialization
LReLU	Leaky Rectified Linear Unit
MLP	Multi-Layer Perceptron
MSE	Mean Squired Error
NLL	Negative Log-Likelihood
RBM	Restricted Boltzmann Machine
RC	Reconstruction Cost
MLP	multilayer perceptron
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
ssRBM	spike and slab Restricted Boltzmann Machine
VAE	Variational Auto-Encoder

Acknowledgments

For the ancestors who paved the path before me, upon whose shoulders I stand. This is also dedicated to my parents Parviz and Effat, to my siblings Ehsan and Sara, and to my nephew and niece Parsa and Aida.

I would like to express my deepest gratitude to my supervisors Prof. Aaron Courville and Prof. Yoshua Bengio for their unwavering support, collegiality, and mentorship throughout this thesis.

I would like to extend my thanks to my co-authors Dr. Philemon Brakel whose for his encouragement, support, guidance, and knowledge; and Mr. Jim Fan for his hard-work, willingness to help, and his unsurpassed experience.

I must also acknowledge great help and support from my friends and colleagues, in alphabetical order Amirbahador Gahroosi, Arnaud Bergeron, Caglar Gulchehre, César Laurent, Dmitry Serdiuk, Faruk Ahmed, Francesco Visin, Frédéric Bastien, Jörg Bornschein, Kyle Kastner, Martin Arjovsky, Mohamed Ishmael Belghazi, Myriam Côté, Negar Rostamzadeh, Pascal Lamblin, Reyhane Askari Hemmat, Saizheng Zhang, Samira Shabanian, Seyed Mohammad Mehdi Ahmadpanah, Sina Honari, Soroush Mehri, Tegan Maharaj, and Ying Zhang.

Finally, the work reported in this thesis would not have been possible without the financial support from: NSERC, Calcul Quebec, Compute Canada, the Canada Research Chairs and CIFAR.

1 Artificial Neural Networks

1.1 Artificial Neural Networks

Inspired by the human brain, Artificial Neural Networks (also called simply *neural networks*) are data processing systems composed of many small processing units. In an analogy to the brain, each of these small processing units is called an *artificial neuron*. Usually, lots of artificial neurons are connected to one another in a hierarchical layered structure. Moreover, to mathematically model the firing rate [1] of each neuron, an *activation function* is used on the output of each neuron. In the next four sub-sections, we introduce the mathematical formulations of ANNs and touch the biological inspirations behind these models.

1.1.1 Artificial neuron

An artificial neuron is a simple function from one or more inputs to a single output. Consider a set of inputs $\mathbf{x} = \{x_1, x_2, ..., x_d\}$ containing d input scalars. A set of d scalar weights $\mathbf{w} = \{w_1, w_2, ..., w_d\}$ are assigned to each input in addition to a single scalar bias term b. Formally, an artificial neuron $h(\mathbf{x})$ is defined as follows,

$$h(\mathbf{x}) = g(\sum_{i=1}^{d} w_i x_i + b), \tag{1.1}$$

in which $g(.)$ is a nonlinear function called the *activation function*. Consequently, we refer to the term $\sum_i w_i x_i + b$ as *the pre-activation*. For simplicity, the summation term can be written in vector multiplication,

$$h(\mathbf{x}) = g(\mathbf{w}^T \mathbf{x} + b). \tag{1.2}$$

1. See more about firing rate in section 1.1.4

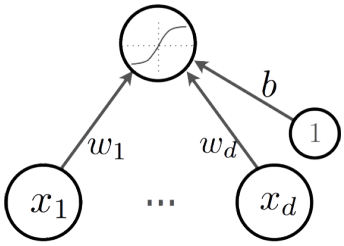

Figure 1.1 – A graphical illustration of an artificial neuron. The input is the vector $\mathbf{x} = \{x_1, x_2, ..., x_d\}$ to which a weight vector $\mathbf{w} = \{w_1, w_2, ..., w_d\}$ and a bias term b is assigned. (Figure adapted from Hugo Larochelle's slides)

Note that for a single set of inputs, $\mathbf{x}$ is a vector of size $d \times 1$ while in the case of N sets of inputs, $\mathbf{x}$ is a matrix of size $d \times N$.

1.1.2 Activation function

In Equation 1.1 the pre-activation is simply a linear weighted sum. In order to make the function from input to output nonlinear, the activation function is applied on the pre-activation. Among different activation functions, here we introduce four of them.

Sigmoid activation function is an element-wise function that ranges between 0 and 1. The output of this function can be interpreted as the probability of activation. Denoting the pre-activation as z, this function is defined as follows,

$$g(z) = \frac{1}{1 + e^{-z}}. \tag{1.3}$$

Rectifier Linear Unit (ReLU) is also an element-wise function which for negative inputs, is simply off (output is zero), while for positive inputs, the output is the same as the input. This activation function is one of the most common activation functions. We can simply define this function as follows,

$$g(z) = \begin{cases} 0 & z \leq 0 \\ z & z > 0 \end{cases} \tag{1.4}$$

Leaky ReLU (LReLU) (Maas et al., 2013) is an extension of the ReLU activation function in which even in the negative region, the output is a small

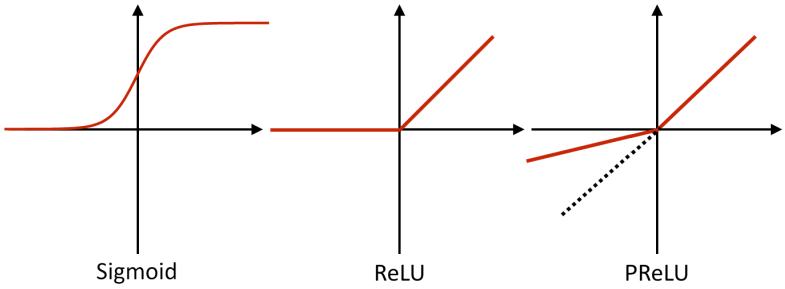

Figure 1.2 – **(a)** the Sigmoid activation function, **(b)** the Rectifier Linear Unit, and **(c)** the Leaky Rectifier Linear Unit.

negative value. The intuition behind this activation function is that empirically, optimization and training benefit from having a small amount of gradient in the negative region. Compare this to the ReLU, where in negative region the neuron's output is zero and there is a gradient of zero in that region. LReLU is defined as follows:

$$g(z) = \begin{cases} \alpha * z & z \leq 0 \\ z & z > 0 \end{cases} \tag{1.5}$$

in which the term α denotes the slope of output in the negative region. Typically, α is set to have a value between 0 and 0.2.

Softmax is another rather different type of activation function, which normalizes a set of pre-activations in such a way that each can be interpreted as a probability. Usually, in classification problems, if there are only two classes, a single Sigmoid unit is used. However, if there are more than two classes, the Softmax is used. Considering $\mathbf{z}$ as a vector of C pre-activations, the Softmax over these C classes is defined as follows,

$$g(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{i=1}^{C} e^i}. \tag{1.6}$$

A graphical depiction of the first three activation functions is shown in Figure 1.2. Note that since the Softmax is applied on more than two numbers, it is not as visualizable as others.

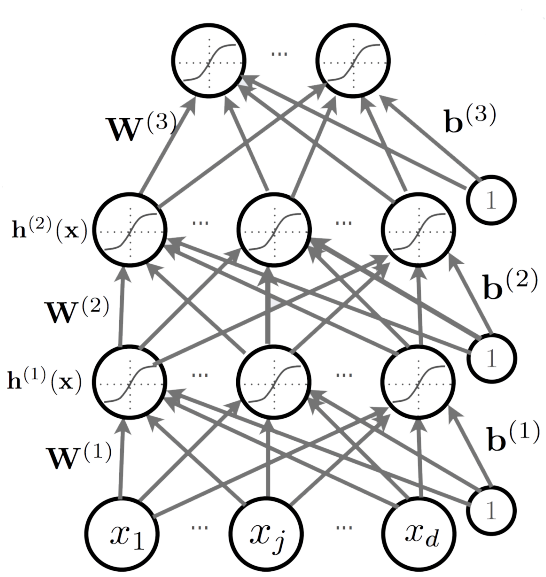

Figure 1.3 – A Three layer neural network. The matrix $\mathbf{W}^{(k)}$ connects the $(k-1)^{th}$ layer to the k^{th} layer and therefore $\mathbf{W}^{(k)} \in \mathbb{R}^{D^k \times D^{k-1}}$ and $\mathbf{b}^{(k)} \in \mathbb{R}^{D^k}$. After each linear transformation (weight multiplication and bias addition), an activation function is applied. (Figure adapted from Hugo Larochelle's slides)

1.1.3 Multilayer neural network

Artificial Neural Networks are usually organized in a layer-wise structure. As shown in figure 1.3, in such a structure, in an individual layer, neurons with different weights and biases [2] are applied on the same input. Then, the output of that layer is the input for the next layer. Formally, a multilayer neural network is defined as follows,

$$\mathbf{h}^{(k)}(\mathbf{x}) = g(\mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})), \tag{1.7}$$

in which, $\mathbf{h}^{(k)}(\mathbf{x})$ is the nonlinear output of k^{th} layer and $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$. Note that if $\mathbf{h}^{(k-1)}(\mathbf{x}) \in \mathbb{R}^{D^{k-1}}$ and $\mathbf{h}^{(k)}(\mathbf{x}) \in \mathbb{R}^{D^k}$, then $\mathbf{W}^{(k)} \in \mathbb{R}^{D^k \times D^{k-1}}$ and $\mathbf{b}^{(k)} \in \mathbb{R}^{D^k}$.

To sum up, a multilayer neural network can be seen as a complicated function from inputs to outputs, composed of many small functions. According to the universal approximation theorem Hornik et al. (1989), multilayer feedforward networks are capable of approximating any measurable function to any desired degree

2. We refer to the weights $\mathbf{W}$'s and biases $\mathbf{b}$'s as parameters.

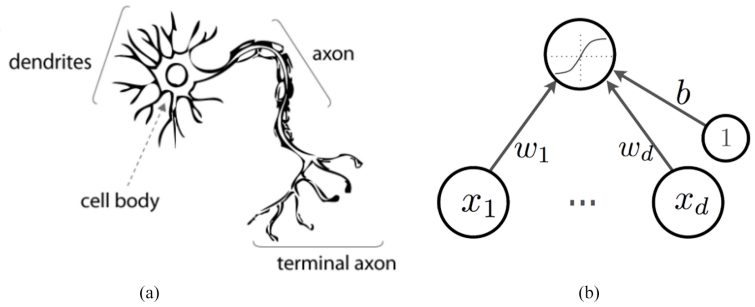

Figure 1.4 – **(a)** a biological neuron: each neuron receives some inputs through the input ports called dendrites and sends some outputs through the axons. **(b)** the abstract artificial model of a biological neuron: the activation corresponds to the firing rate, the weights correspond to the connection strength between two neurons, and the activation function and bias term correspond to the threshold of firing.

of accuracy given sufficient number of layers and neurons per each layer. However, finding the appropriate parameters $\mathbf{W}$'s and $\mathbf{b}$'s remains a challenging problem. In section 1.2, we introduce the current methods for training such architectures.

1.1.4 Biological inspiration

Biological neural networks, such as the human brain, are made of billions of biological neurons. Biological neurons communicate by sending and receiving electro-chemical signals. In the most popular simple models of neuron behaviour, each neuron has some inputs and outputs which can be either on or off. Each transition between being on or off is called a *spike* and the number of spikes per unit of time is called *firing rate*. A neuron fires (outputs a spike) when it receives input spikes above a certain threshold. With these simple models, it is generally considered that when two adjacent neurons fire together, the weight between the two is strengthened and vice versa.

The artificial neuron as described in section 1.1.1 is an abstract model of a biological neuron, in which the weight between two adjacent neurons is modeled as a scalar and the firing rate is modeled by the activation function. Figure 1.4 shows a single biological neuron and its abstract artificial model.

Biologically, human visual cortex has a layer-wise structure, which has been an inspiration for multilayer artificial neural networks. As shown in Figure 1.5, when light hits our eyes, retinal neurons process the light into electro-chemical signals and send them (possibly via other layers) to Lateral Geniculate Nucleus (LGN)

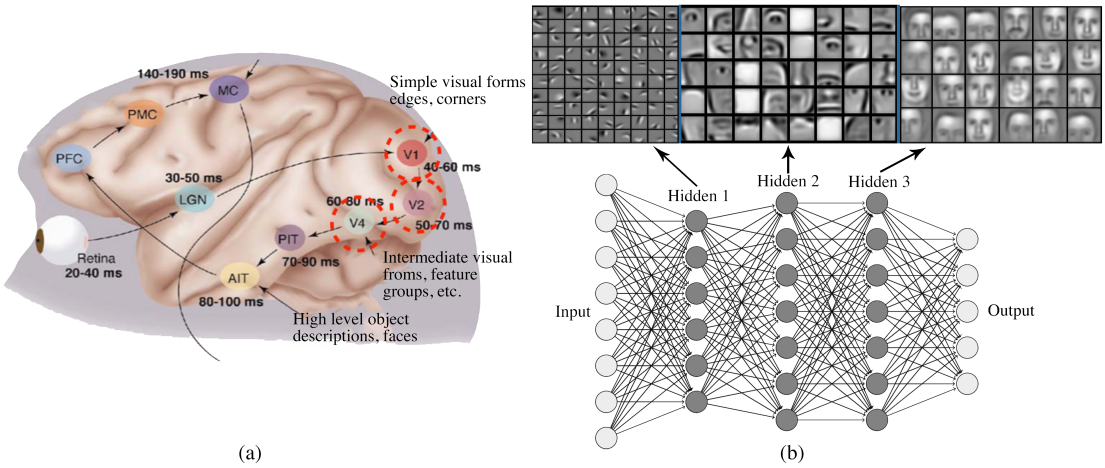

Figure 1.5 – **(a)** A depiction of human visual cortex (Thorpe & Fabre-Thorpe, 2001). **(b)** A multilayer artificial neural network imitating the visual cortex Lee et al. (2009), and a visualization of filters learned by a trained multilayer artificial neural network, showing that it is also capable of detecting edges, patterns, and objects in different layers.

and from LGN to other layers. Thorpe & Fabre-Thorpe (2001) shows that the V_1 is responsible for detecting edges and corners, V_4 is responsible for detecting intermediate visual forms, and finally AIT is responsible for high-level abstract object descriptions.

1.2 Training Artificial Neural Networks

In the previous section, we introduced the structure of artificial neural networks. We also defined two sets of parameters: weights and biases. In order that given a certain input to the network, model outputs a certain output, the parameters need to be adapted. The process of adapting the parameters is called *training*. In the following subsections, we introduce methods in order to optimize the parameters with respect to a predifined loss function.

1.2.1 Cost Functions

In optimization problems, *cost function* or *loss function* or *objective function* is a function from a set of variables or values to a single real number. This single

real number is called the "cost". Usually the objective of training procedure is to minimize the cost (or maximize its negative).

To design an empirical cost function, consider a network with parameters θ from the input $\mathbf{x}$ to the output $\hat{\mathbf{y}}$. The objective is to minimize the loss between the output $\hat{\mathbf{y}}$ and the target $\mathbf{y}$. Having T pairs of $(\mathbf{x}^{(t)}, \mathbf{y}^{(t)})$ and $\hat{\mathbf{y}}^{(t)} = f(\mathbf{x}^{(t)}; \theta)$ the cost function is defined as follows:

$$L(\mathbf{x}, \theta) = \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}), \tag{1.8}$$

in which, depending on the task, $l(f(\mathbf{x}^{(t)}; \theta), \mathbf{y})$ might be Mean Square Error (MSE), negative log-likelihood (nll), or other differentiable functions.

In a classification task with C classes, the value of each output neuron is interpreted as the probability of that specific class, i.e., $f(\mathbf{x})_c = Pr(y = c|\mathbf{x})$. In such a task, the function $l(.,.)$ can be the negative log-likelihood:

$$l(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}) = -\sum_{c} 1_{y=c} \log f(\mathbf{x})_c = -\log f(\mathbf{x})_y, \tag{1.9}$$

in which the log is used for numerical stability and also for mathematical simplicity.

In the regression task, the target $\mathbf{y}$ is a vector of real values, the cost function is often the MSE:

$$l(f(\mathbf{x}^{(t)}; \theta), \mathbf{y}) = ||f(\mathbf{x}^{(t)}; \theta) - \mathbf{y}||_F^2, \tag{1.10}$$

where $||.||_F^2$ is the Frobenius norm. Frobenius norm of a vector V with n elements is defined as follows:

$$||A||_F^2 = \sqrt{\sum_{i=1}^{n} a_i^2}. \tag{1.11}$$

1.2.2 Gradient Based Optimization

Artificial neural networks are typically trained using gradient based optimization methods and specifically, using the Gradient Descent algorithm and its variants. Gradient Descent can be seen as approximating a function by its first-order Taylor series. Gradient Descent finds a local minimum of a function by taking small

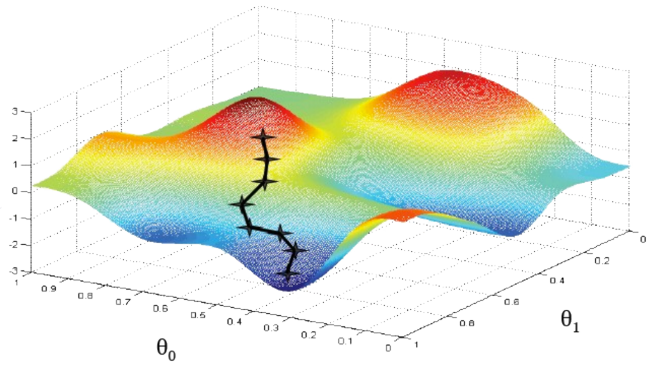

Figure 1.6 – A graphical depiction of the gradient descent algorithm. In this case, on two axes, there are values of two parameters and on the z axis the value of the cost function is visualized. (Figure adapted from Andrew Ng's slides.)

steps in the direction of the gradient proportional to its magnitude. Therefore, gradient descent is an iterative process that at each iteration updates the parameters using the following update rule,

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla_{\theta_t} L(\mathbf{x}, \theta_t), \tag{1.12}$$

where η is called the *learning rate*.

In simple words, the gradient descent algorithm can be seen as a hiker climbing down a hill to a part with lowest height. Each step of the hiker is determined by the slope of the hill at that specific location. A graphical depiction of the gradient descent algorithm is shown in figure 1.6.

Typically, **Stochastic (mini-batch) Gradient Descent** (SGD) is used rather than Gradient Descent. SGD is a version of Gradient Descent in which instead of computing the gradient $\nabla_\theta L(\mathbf{x}, \theta)$ exactly, an estimate of the gradient is used based on one or a few randomly selected examples. Since the examples are randomly selected, the expected value of the gradient is the same as the exact gradient.

1.2.3 Adam Learning Algorithm

In practice, the learning rate is an important hyperparameter that can affect learning significantly. One way to handle this problem is to use learning rates which can adapt throughout the course of learning. In recent years, several algorithms

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 1.7 – A pseudo code for the Adam learning algorithm (Kingma & Ba, 2014). Note that both mean and variance are computed using two moving averages starting at zero. As a result, the moving averages are biased towards zero which are corrected.

with adaptive learning rate have been proposed. In this section, we introduce *Adam* (Kingma & Ba, 2014) which is used in all of our experiments.

The *Adam* algorithm has its name derived from "adaptive moments". It adapts the learning rate of each parameter by scaling them. The scaling factor is proportional to an exponentially weighted moving average over the accumulated gradient. The pseudo code of this algorithm is shown in Figure 1.7.

Other than the global learning rate, there are two other hyperparameters ρ_1 and ρ_2. In practice, if the number of mini-batches is N, the best value for ρ_2 is $1 - \frac{1}{N}$.

1.2.4 Backpropagation Algorithm

In the previous sections, we showed how to train neural network parameters using gradients of their parameters. In this section, we introduce a well-known algorithm for computing the gradients in an efficient way. In a multi-layer network, backpropagation uses the chain rule to iteratively compute the gradients. Obviously, in order to be able to use backpropagation, both activations and pre-activations must be differentiable.

Each step of the backpropagation contains one forward and one backward paths. Forward path means feeding the network input, computing the pre-activations and

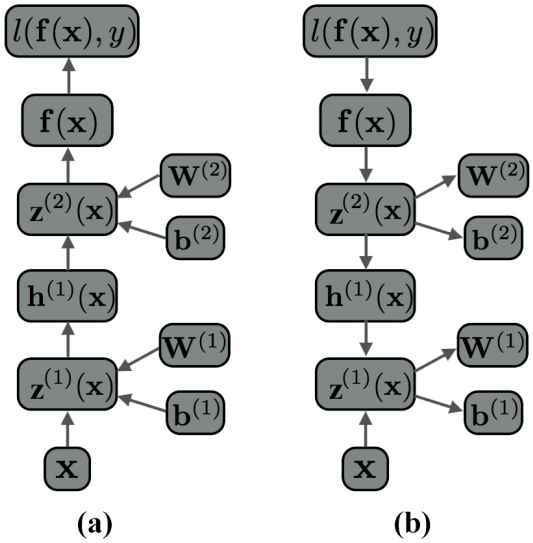

Figure 1.8 – (a) The forward path and (b) the backward path in backpropagation algorithm on a two-layer neural network. Note that in the backward path, in order to be able to go through each module, it must be differentiable. (Figure adapted from Hugo Larochelle's slides.)

activations, and finally computing the error (the cost function). Similarly, the backward path amounts to propagation of errors on activations and then pre-acticated of each layer. Figure 1.8 depicts the forward and backward paths using a flow graph.

Formally, consider the network in Figure 1.8 and the cost function as the negative log-likelihood as described in sub-section 1.2.1. Consistent with our previous notation, $\mathbf{z}^{(l)}$ and $\mathbf{h}^{(l)}$ are the pre-activations and activations at layer l, respectively.

The gradient of the loss w.r.t. the pre-activation at layer 2 is,

$$\nabla_{z^{(2)}(\mathbf{x})} - \log f(\mathbf{x})_y = -(\mathbf{e}(y) - f(\mathbf{x})), \tag{1.13}$$

in which $\mathbf{e}(y)$ is a one-hot representation that all the elements are zero except the element in index y which is one. Using chain rule, since $\nabla_{\mathbf{W}^{(2)}} z^{(2)}(\mathbf{x}) = \mathbf{h}^{(1)}(\mathbf{x})$ and $\nabla_{\mathbf{b}^{(2)}} z^{(2)}(\mathbf{x}) = \mathbf{1}$, the gradients w.r.t. the parameters can be derived as follows,

$$\nabla_{\mathbf{W}^{(2)}} - \log f(\mathbf{x})_y = \big(\nabla_{z^{(2)}(\mathbf{x})} - \log f(\mathbf{x})_y\big)\mathbf{h}^{(1)}(\mathbf{x})^T, \tag{1.14}$$

$$\nabla_{\mathbf{b}^{(2)}} - \log f(\mathbf{x})_y = \nabla_{z^{(2)}(\mathbf{x})} - \log f(\mathbf{x})_y. \tag{1.15}$$

To back-propagate the gradient to the next layer (specifically, the next pre-activation),

we have,

$$\nabla_{\mathbf{h}^{(1)}(\mathbf{x})} - \log f(\mathbf{x})_y = \mathbf{W}^{(2)^T}\big(\nabla_{\mathbf{z}^{(2)}(\mathbf{x})} - \log f(\mathbf{x})_y\big), \tag{1.16}$$

$$\nabla_{\mathbf{z}^{(1)}(\mathbf{x})} - \log f(\mathbf{x})_y = \big(\nabla_{\mathbf{h}^{(1)}(\mathbf{x})} - \log f(\mathbf{x})_y\big) \odot g'(\mathbf{z}^{(1)}(\mathbf{x})), \tag{1.17}$$

in which $\odot$ is an element-wise multiplication and $g'(.)$ is the derivative of the activation function. Having $\nabla_{\mathbf{z}^{(1)}(\mathbf{x})} - \log f(\mathbf{x})_y$, gradients of parameters in the first layer can be computed in a similar way.

1.2.5 Faster Training using Batch Normalization

During training deep neural networks, as a result of changing parameters over the course of learning, the distributions of representations at each layer change. This change in distribution which makes the training procedure slower is known as *Internal Covariance Shift* problem. It is hypothesized that reducing the Internal Covariance Shift helps both optimization and generalization (Ioffe & Szegedy, 2015).

One way to reduce the Internal Covariance Shift is a recently proposed method named *Batch Normalization* (Ioffe & Szegedy, 2015). Batch Normalization standardize the pre-activations using the sample mean and sample variance over the current mini-batch. Consider a mini-batch of pre-activations $\mathbf{z} \in \mathbb{R}^{M \times k}$ in which M is the number of examples in the mini-batch and k is the number of features. The sample mean and sample variance vectors are computed as follows,

$$\bar{\mathbf{z}} = \frac{1}{M}\sum_{i=1}^{M} \mathbf{z}_{i.}, \tag{1.18}$$

$$\sigma^2 = \frac{1}{M}\sum_{i=1}^{M} (\mathbf{z}_{i.} - \bar{\mathbf{z}})^T(\mathbf{z}_{i.} - \bar{\mathbf{z}}). \tag{1.19}$$

Therefore, both $\bar{\mathbf{z}}$ and σ^2 are vectors in $\mathbb{R}^k$. Using these statistics, we can apply normalization on $\mathbf{z}$,

$$\hat{\mathbf{z}} = \frac{\mathbf{z} - \bar{\mathbf{z}}}{\sqrt{\sigma^2 + \epsilon}}. \tag{1.20}$$

in which ϵ is a small constant for numerical stability. However, such normalization

reduces the representational power of each layer. To resolve this, two extra learnable parameters β and γ are added and multiplied, respectively. Consequently, if we denote a batch normalization layer as $BN(.)$, the formulation is as following,

$$BN(\mathbf{z}) = \gamma\hat{\mathbf{z}} + \beta. \tag{1.21}$$

Note that because of β, the bias term of the linear transformation can be removed. Moreover, in the case of activation functions like ReLU and LReLU where only the sign of the input matters and not the scale, the γ term is usually removed. It is worth mentioning that during test time, we compute $\bar{\mathbf{z}}$ and σ^2 over the training set.

2 Representation Learning

The success of many Machine Learning algorithms depends on data representation. For example, a feature representation that successfully separates distinct classes can lead to perfect learning via simply a linear classifier in this representation space. In the past decades, it has been the norm for human experts to design task-specific representations using domain-specific knowledge, prior assumptions, or at times, simply trial-and-error. Despite the reasonable success of this approach for specific tasks, we would ideally like to develop end-to-end trainable models that learn the best feature representation for the task on its own. Over the last decade, an explosion in the amount of the available data combined with increasingly powerful computational resources has resulted in representation learning [1] methods providing significant gains in performance across a wide range of tasks. For the task of object recognition, representation learning methods have achieved performance comparable to that of humans on the standard ImageNet dataset (Krizhevsky et al., 2012) (He et al., 2015). In the area of Natural Language Processing, for different tasks such as Machine Translation (Bahdanau et al., 2014), Sentiment Analysis (Glorot et al., 2011), Language Modeling (Graves, 2013), representation learning based algorithms currently hold state-of-the-art performance.

In this chapter, we first discuss two typical approaches for representation learning: 1) Supervised Learning and 2) Unsupervised Learning. Secondly, we discuss how to combine these two approaches. In the following section, we start by introducing the idea of disentangling factors of variation.

1. Usually, both terms *representation learning* and *deep learning* are used interchangeably.

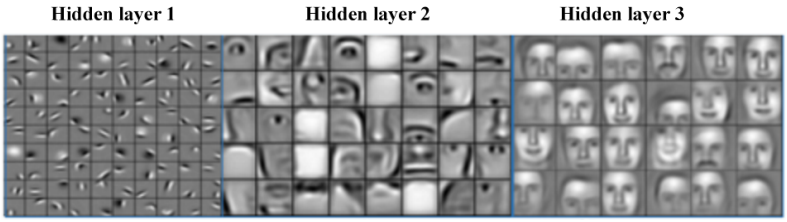

Figure 2.1 – Learned features in a deep neural network. As we go deeper into the network, the network has learned to extract increasingly higher levels of abstraction from raw pixel input data. (Figure adapted from Lee et al. (2009))

2.1 Disentangling Factors of Variation

All sensory data that we receive, such as rays of light striking our retina (a matrix of pixels for computers), are generated as a result of interactions between many latent factors of variation. Different factors such as pose, face expression, skin color, and illumination interact in real world and the result is a matrix of pixels. Disentangling amounts to the process of teasing apart these factors of variations and extracting high-level latent features from low-level sensory data.

As we discussed, the low-level, high-dimensional sensory data is provided to a neural network through the input layer and it activates neurons in the subsequent layers. The activations in each layer correspond to pattern detectors, such that as we proceed deeper into the network, the patterns get more abstract. For example, for a typical computer vision task, the first layer represents pixels, the next layer learns to represent edges and patches, and the upper layers learn to represent high-level scene components such as objects (Lee et al., 2009). Figure 2.1 provides a visualization for the pattern detectors at each layer.

The fundamental principle of deep learning is that given enough data, a sufficiently deep neural network is able to extract high-level features. Depending on whether datapoints are labeled or not, we have two different categories of learning tasks: supervised and unsupervised learning. Consider a computer vision task in which the data contain only pixels and no label information. In such a scenario, an unsupervised neural network can discover that neighboring pixels have strong correlations with one another. Subsequently, other higher-level features can be built on top of the discovered feature representation. On the other hand, if labels are provided as well, they can be used to guide the network towards representations

that are more task-specific. In the following two sections, we discuss supervised and unsupervised learning in more detail.

2.2 Supervised Learning

In supervised learning, both the input and the desired output (or the target) are provided during training. Training amounts to finding a mapping between the input and the target.

Supervised learning problems are usually categorized into *regression* and *classification* tasks. In a regression problem the target is a continuous variable, and for classification the target is categorical and discrete. In neural network applications, the most common supervised learning problem is classification. Object detection, activity detection and handwritten digit recognition are all examples of classification [2].

From a probabilistic point of view, the goal of supervised learning is to learn a conditional probability distribution that can be used for making predictions. Such a network is usually trained using a cross entropy cost function [3]. Consider a multilayer neural network $f(.)$ fed with a single input vector, $\mathbf{x}$, that predicts the number of classes, C, by modeling the following conditional distribution,

$$Pr(y = c|\mathbf{x}) = f(\mathbf{x})_c = \hat{\mathbf{y}}_c. \tag{2.1}$$

As described in section 1.2.1, for N training examples, the total cost function is defined as follows,

$$Cost = -\frac{1}{N}\sum_{n=1}^{N} \log Pr(\hat{\mathbf{y}}^{(n)} = \mathbf{y}^{(n)}|\mathbf{x}^{(n)}), \tag{2.2}$$

where $\mathbf{y}$ is a one-hot vector with zeros everywhere except the c^{th} element, which is

2. However, for all of these examples other types of learning may also be used, for example unsupervised pretraining.

3. Since in classification the target is categorical, the cost function is called *categorical cross entropy*.

one. Such a network can be trained using Stochastic Gradient Decent and back-propagation as described in the previous chapter.

Although deep supervised learning has been achieving impressive results, purely supervised learning requires a huge amount of labeled data for deep models to work well. Besides, sometimes, the learned representations in a supervised network are optimized for a specific task and may not be transferable to other tasks. Since unsupervised learning does not require label information, unsupervised learning is of interest, since unlabeled data is cheap and easy to come by these days.

2.3 Unsupervised Learning

In unsupervised learning, the training data consists of a set of input datapoints, $\mathbf{x}$, without any corresponding target values. The principal idea behind unsupervised learning is that only the raw input without any other information is sufficient for the model to learn a meaningful representation of the data. In conventional machine learning, *clustering* is a common example of unsupervised learning in which the task is to group examples that are *similar*[4] to each other.

In the deep learning literature, *unsupervised feature extraction*, *density estimation*, and *manifold learning* are the most common methods of deep unsupervised learning. Deep learning and deep neural networks seem to be a great candidate for feature extraction, following upon the intuition behind the compositionality assumption that assumes more abstract features are made out of less abstract ones (Bengio et al., 2013a). Density estimation also amounts to adapting model parameters in order to recover Pr_{data} given a training set. In manifold learning, the idea is that the complicated distribution Pr_{data} can be modeled with a simple distribution $Pr_{\mathbf{z}}$ in which $\mathbf{z}$ is a latent variable that lies on a low-dimensional space.

Since unsupervised learning does not require label information, massive amounts of readily available unlabeled data can be used to train such models. Different models such as *Restricted Boltzmann Machine* (RBM), *Deep Belief Network* (DBN) (Hinton et al., 2006), *spike and slab Restricted Boltzmann Machine* (ss-RBM) (Courville et al., 2011), Stacked Autoencoders (Hinton & Salakhutdinov,

4. Some quantified measure of similarity is used.

2006), and Sparse Coding are all examples of unsupervised deep learning. Among these, we introduce, in the next subsections, two unsupervised models that have achieved successful results in the last decade.

2.3.1 Auto-Encoders

Auto-Encoders (Hinton & Salakhutdinov, 2006) are models that map (encode) the data in input space to another hidden space and then map the hidden representation back (decode) to the input space (reconstruction). The objective is to train a model in a way that has the lowest reconstruction error on test examples. Formally, given a single input vector $\mathbf{x}$ and the encoder function $f_\theta(.)$, we have,

$$\mathbf{h} = f_\theta(\mathbf{x}), \tag{2.3}$$

where θ is the set of parameters and $\mathbf{h}$ is the hidden representation. Similarly, the decoder has the following form,

$$\hat{\mathbf{x}} = g_\phi(\mathbf{h}). \tag{2.4}$$

To measure the discrepancy between the reconstruction $\hat{\mathbf{x}}$ and the data $\mathbf{x}$, in a basic auto-encoder the cost function is usually the Mean Square Error (MSE),

$$Cost = ||\hat{\mathbf{x}} - \mathbf{x}||_F^2. \tag{2.5}$$

The two encoder and decoder functions are usually parameterized by neural networks. Figure 2.2 (a) is a graphical depiction of a standard auto-encoder.

2.3.2 Denoising Auto-Encoders

In the case of *overcomplete* auto-encoders in which the dimensionality of hidden representation is larger than the dimensionality of input data, the auto-encoder may "cheat" and learn an identity mapping that leads to zero reconstruction cost. One way to prevent the model from learning a trivial mapping is to artificially corrupt the input by adding noise to it before feeding it to the model. The model must now reconstruct the uncorrupted input from its noisy version. The idea is that the network must learn the data structure in order to be able to undo the corruption

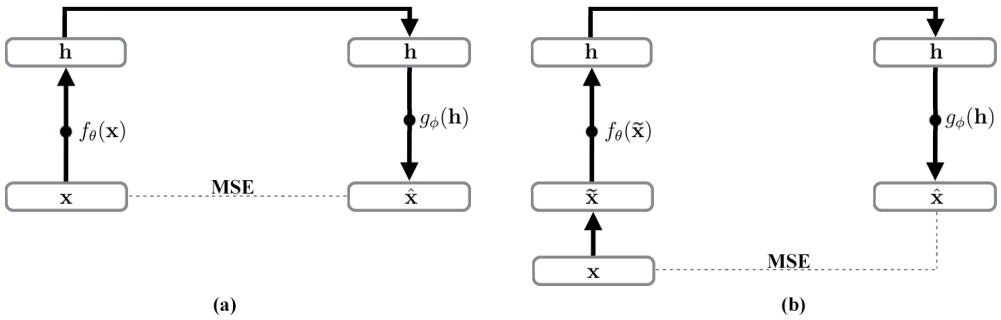

Figure 2.2 – (a) An auto-encoder: the input to the network is $\mathbf{x}$ and the output is the reconstruction $\hat{\mathbf{x}}$. Note that each of $f_\theta(.)$ and $g_\phi(.)$ can be deep neural networks. (b) A denoising auto-encoder: the input to the network is noisy, but the reconstruction $\hat{\mathbf{x}}$ is compared to the uncorrupted data.

process. Such an extension of auto-encoders is called denoising auto-encoder (DAE) (Vincent et al., 2010b).

In practice, the denoising auto-encoder leads to qualitatively better features. Besides, a better classification performance can be achieved by using features from a denoising auto-encoder rather than those from an standard auto-encoder.

From a manifold learning point of view, in the hidden representation space, all data points lie on a low-dimensional manifold. When a corrupted data point is fed to the network, it lies somewhere farther from the data manifold. Subsequently the model must learn the probability distribution $Pr(\mathbf{x}|\hat{\mathbf{x}})$ to project the noisy data point back to the manifold. A graphical illustration of this process is shown in Figure 2.3.

2.4 Combining Supervised and Unsupervised Learning

Labeling data sets is typically a costly task and in many settings there are far more unlabeled examples than labeled ones. Supervised learning algorithms on small amount of labeled data can result in severe overfitting. As a result, it is desirable to take advantage of huge amount of unlabeled data to learn representations which are useful for the supervised task.

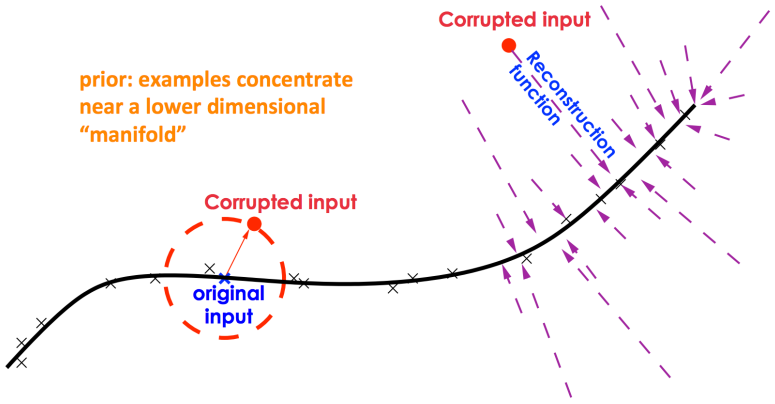

Figure 2.3 – A visual illustration of the data manifold in low-dimensional hidden space. A corrupted input lies far from the data manifold and the reconstruction function projects corrupted inputs back. (Figure adapted from Vincent et al. (2010b))

Semi-supervised learning aims to improve the performance on some supervised learning problem by using information obtained from both labeled and unlabeled examples. Since the recent success of deep learning methods has mainly relied on supervised learning based on very large labeled datasets, it is interesting to explore semi-supervised deep learning approaches to extend the reach of deep learning to these settings.

Historically, unsupervised learning played an important role in the first wave of deep learning (Hinton et al., 2006; Vincent et al., 2008; Bengio, 2009). Unsupervised pre-training of neural networks was used to initialize parameters of models for supervised training, a process referred to as *greedy layer-wise unsupervised pretraining*. It's considered a *greedy* method because the layers are learned one at a time without consideration of what is being trained next [5]. Consequently, a natural next step is to investigate how ideas inspired by Restricted Boltzmann Machine training and regularized autoencoders can be used for semi-supervised learning. Examples of approaches based on such ideas are the discriminative RBM (Larochelle & Bengio, 2008) and a deep architecture based on semi-supervised autoencoders that was used for document classification (Ranzato & Szummer, 2008).

Recent examples of semi-supervised deep learning are the semi-supervised Variational Autoencoder (Kingma et al., 2014) and the Ladder Network (Rasmus et al., 2015) which obtained state-of-the-art results (1.13% error) on the MNIST hand-

5. There is no global objective being trained.

written digits classification benchmark using just 100 labeled training examples.

The Ladder Network adds an unsupervised component to the supervised learning objective of a deep feedforward network by treating this network as part of a deep stack of denoising autoencoders or DAEs (Vincent et al., 2010a) that learns to reconstruct each layer (including the input) based on a corrupted version of it, using feedback from upper levels. The term "ladder" refers to how this architecture extends the stacked DAE in the way the feedback paths are formed.

3 Prologue to the Article

Deconstructing the Ladder Network Architecture. Mohammad Pezeshki, Linxi Fan, Philémon Brakel, Aaron Courville, and Yoshua Bengio. Proceedings of the 33rd International Conference on Machine Learning (ICML), 2016.

Personal Contribution. The underlying idea of performing a deconstructive study of the Ladder Architecture via an ablation study in a semi-supervised setup was mine. Philémon Brakel and I designed different variants of the architecture. Then, I conducted the experiments for the variants derived by removal of individual components, and Linxi Fan conducted the experiments derived by replacement. I implemented the majority of the code in Theano (Bergstra et al., 2010; Bastien et al., 2012), Blocks and Fuel (van Merriënboer et al., 2015), based on code from Rasmus et al. (2015); Valpola (2014). I contributed significantly to the writing of the paper, with valuable inputs from my supervisors Yoshua Bengio and Aaron Courville. My co-authors Philémon Brakel and Linxi Fan also reviewed and rewrote some parts.

4 Deconstructing the Ladder Network Architecture

4.1 Introduction

Labeling data sets is typically a costly task and in many settings there are far more unlabeled examples than labeled ones. Semi-supervised learning aims to improve the performance on some supervised learning problems by using information obtained from both labeled and unlabeled examples. Since the recent success of deep learning methods has mainly relied on supervised learning based on very large labeled datasets, it is interesting to explore semi-supervised deep learning approaches to extend the reach of deep learning to these settings.

Since unsupervised methods for pre-training layers of neural networks were an essential part of the first wave of deep learning methods (Hinton et al., 2006; Vincent et al., 2008; Bengio, 2009), a natural next step is to investigate how ideas inspired by Restricted Boltzmann Machine training and regularized autoencoders can be used for semi-supervised learning. Examples of approaches based on such ideas are the discriminative RBM (Larochelle & Bengio, 2008) and a deep architecture based on semi-supervised autoencoders that was used for document classification (Ranzato & Szummer, 2008). More recent examples of approaches for semi-supervised deep learning are the semi-supervised Variational Autoencoder (Kingma et al., 2014) and the Ladder Network (Rasmus et al., 2015) which obtained state of the art results (1.13% error) on the MNIST handwritten digits classification benchmark using just 100 labeled training examples.

The Ladder Network adds an unsupervised component to the supervised learning objective of a deep feedforward network by treating this network as part of a deep stack of denoising autoencoders or DAEs (Vincent et al., 2010a) that learns to reconstruct each layer (including the input) based on a corrupted version of it, using feedback from upper levels. The term 'ladder' refers to how this architecture extends the stacked DAE in the way the feedback paths are formed.

This paper is focusing on the design choices that lead to the Ladder Network's superior performance and tries to disentangle them empirically. We identify some general properties of the model that make it different from standard feedforward networks and compare various architectures to identify those properties and design choices that are the most essential for obtaining good performance. While the authors of the Ladder Network paper explored some variants of their model already, we provide a thorough comparison of a large number of architectures controlling for both hyperparameter settings and data set selection. Finally, we also introduce a variant of the Ladder Network that yields new state-of-the-art results for the Permutation-Invariant MNIST classification task in both semi- and fully-supervised settings.

4.2 The Ladder Network Architecture

In this section, we describe the Ladder Network Architecture [1]. Consider a dataset with N labeled examples $(x(1), y^*(1)), (x(2), y^*(2)), ..., (x(N), y^*(N))$ and M unlabeled examples $x(N+1), x(N+2), ..., x(N+M)$ where $M \gg N$. The objective is to learn a function that models $P(y|x)$ by using both the labeled examples and the large quantity of unlabeled examples. In the case of the Ladder Network, this function is a deep Denoising Auto Encoder (DAE) in which noise is injected into all hidden layers and the objective function is a weighted sum of the supervised Cross Entropy cost on the top of the encoder and the unsupervised denoising Square Error costs at each layer of the decoder. Since all layers are corrupted by noise, another encoder path with shared parameters is responsible for providing the clean reconstruction targets, i.e. the noiseless hidden activations (See Figure 4.1).

Through lateral skip connections, each layer of the noisy encoder is connected to its corresponding layer in the decoder. This enables the higher layer features to focus on more abstract and task-specific features. Hence, at each layer of the decoder, two signals, one from the layer above and the other from the corresponding layer in the encoder are combined.

1. Please refer to (Rasmus et al., 2015; Valpola, 2014) for more detailed explanation of the Ladder Network architecture.

Formally, the Ladder Network is defined as follows:

$$\tilde{x}, \tilde{z}^{(1)}, ..., \tilde{z}^{(L)}, \tilde{y} = \text{Encoder}_{noisy}(x), \tag{4.1}$$

$$x, z^{(1)}, ..., z^{(L)}, y = \text{Encoder}_{clean}(x), \tag{4.2}$$

$$\hat{x}, \hat{z}^{(1)}, ..., \hat{z}^{(L)} = \text{Decoder}(\tilde{z}^{(1)}, ..., \tilde{z}^{(L)}), \tag{4.3}$$

where Encoder and Decoder can be replaced by any multi-layer architecture such as a multi-layer perceptron in this case. The variables x, y, and $\tilde{y}$ are the input, the noiseless output, and the noisy output respectively. The variables $z^{(l)}$, $\tilde{z}^{(l)}$, and $\hat{z}^{(l)}$ are the hidden representation, its noisy version, and its reconstructed version at layer l. The objective function is a weighted sum of supervised (Cross Entropy) and unsupervised costs (Reconstruction costs).

$$Cost = -\Sigma_{n=1}^{N} \log P(\tilde{y}_{(n)} = y^*(n)|x_{(n)}) + \Sigma_{n=N+1}^{M} \Sigma_{l=1}^{L} \lambda_l \, \text{ReconsCost}(z_{(n)}^{(l)}, \hat{z}_{(n)}^{(l)}). \tag{4.4}$$

in which, y^* is the true target. Note that while the noisy output $\tilde{y}$ is used in the Cross Entropy term, the classification task is performed by the noiseless output y at test time.

In the forward path, individual layers of the encoder are formalized as a linear transformation followed by Batch Normalization (Ioffe & Szegedy, 2015) and then application of a nonlinear activation function:

$$\tilde{z}_{pre}^{(l)} = W^{(l)} \cdot \tilde{h}^{(l-1)}, \tag{4.5}$$

$$\tilde{\mu}^{(l)} = \text{mean}(\tilde{z}_{pre}^{(l)}), \tag{4.6}$$

$$\tilde{\sigma}^{(l)} = \text{stdv}(\tilde{z}_{pre}^{(l)}), \tag{4.7}$$

$$\tilde{z}^{(l)} = \frac{\tilde{z}_{pre}^{(l)} - \tilde{\mu}^{(l)}}{\tilde{\sigma}^{(l)}} + \mathcal{N}(0, \sigma^2), \tag{4.8}$$

$$\tilde{h}^{(l)} = \phi(\gamma^{(l)}(\tilde{z}^{(l)} + \beta^{(l)})), \tag{4.9}$$

where $\tilde{h}^{(l-1)}$ is the post-activation at layer $l-1$ and $W^{(l)}$ is the weight matrix from layer $l-1$ to layer l. Batch Normalization is applied to the pre-normalization $\tilde{z}_{pre}^{(l)}$ using the mini-batch mean $\mu^{(l)}$ and standard deviation $\sigma^{(l)}$. Functions mean and

stdv are defined as follows,

$$\text{mean}(\mathbf{v}) = \frac{1}{N}\sum_{i=1}^{N}\mathbf{v}_i, \tag{4.10}$$

$$\text{stdv}(\mathbf{v}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\mathbf{v}_i - \text{mean}\mathbf{v})^2}. \tag{4.11}$$

The next step is to add Gaussian noise with mean 0 and variance σ^2 to compute pre-activation $\tilde{z}^{(l)}$. The parameters $\beta^{(l)}$ and $\gamma^{(l)}$ are responsible for shifting and scaling before applying the nonlinearity $\phi(\cdot)$. Note that the above equations describe the *noisy* encoder. If we remove noise ($\mathcal{N}(0, \sigma^2)$) and replace $\tilde{h}$ and $\tilde{z}$ with h and z respectively, we will obtain the noiseless version of the encoder.

At each layer of the decoder in the backward path, the signal from the layer $\hat{z}^{(l+1)}$ and the noisy signal $\tilde{z}^{(l)}$ are combined into the reconstruction $\hat{z}^{(l)}$ by the following equations:

$$u_{pre}^{(l+1)} = V^{(l)} \cdot \hat{z}^{(l+1)}, \tag{4.12}$$

$$\mu^{(l+1)} = \text{mean}(u_{pre}^{(l+1)}), \tag{4.13}$$

$$\sigma^{(l+1)} = \text{stdv}(u_{pre}^{(l+1)}), \tag{4.14}$$

$$u^{(l+1)} = \frac{u_{pre}^{(l+1)} - \mu^{(l+1)}}{\sigma^{(l+1)}}, \tag{4.15}$$

$$\hat{z}^{(l)} = g(\tilde{z}^{(l)}, u^{(l+1)}) \tag{4.16}$$

where $V^{(l)}$ is a weight matrix from layer $l+1$ to layer l. We call the function $g(\cdot,\cdot)$ the *combinator function* as it combines the vertical $u^{(l+1)}$ and the lateral $\tilde{z}^{(l)}$ connections in an element-wise fashion. The original Ladder Network proposes the following design for $g(\cdot,\cdot)$, which we call the *vanilla combinator*:

$$g(\tilde{z}^{(l)}, u^{(l+1)}) = b_0 + w_{0z} \odot \tilde{z}^{(l)} + w_{0u} \odot u^{(l+1)} + w_{0zu} \odot \tilde{z}^{(l)} \odot u^{(l+1)} + \tag{4.17}$$

$$w_\sigma \odot \text{Sigmoid}(b_1 + w_{1z} \odot \tilde{z}^{(l)} + w_{1u} \odot u^{(l+1)} + w_{1zu} \odot \tilde{z}^{(l)} \odot u^{(l+1)}), \tag{4.18}$$

where $\odot$ is an element-wise multiplication operator and each per-element weight

is initialized as:

$$\begin{cases} w_{\{0,1\}z} & \leftarrow 1 \\ w_{\{0,1\}u} & \leftarrow 0 \\ w_{\{0,1\}zu}, b_{\{0,1\}} & \leftarrow 0 \\ w_{\sigma} & \leftarrow 1 \end{cases} \tag{4.19}$$

In later sections, we will explore alternative initialization schemes on the vanilla combinator. Finally, the ReconsCost$(z^{(l)}, \hat{z}^{(l)})$ in equation (4.4) is defined as the following:

$$\text{ReconsCost}(z^{(l)}, \hat{z}^{(l)}) = ||\frac{\hat{z}^{(l)} - \mu^{(l)}}{\sigma^{(l)}} - z^{(l)}||^2. \tag{4.20}$$

where $\hat{z}^{(l)}$ is normalized using $\mu^{(l)}$ and $\sigma^{(l)}$ which are the *encoder*'s sample mean and standard deviation statistics of the current mini batch, respectively. The reason for this second normalization is to cancel the effect of unwanted noise introduced by the limited batch size of Batch Normalization.

4.3 Components of the Ladder Network

Now that the precise architecture of the Ladder Network has been described in details, we can identify a couple of important additions to the standard feed-forward neural network architecture that may have a pronounced impact on the performance. A distinction can also be made between those design choices that follow naturally from the motivation of the ladder network as a deep autoencoder and those that are more ad-hoc and task specific.

The most obvious addition is the extra reconstruction cost for every hidden layer and the input layer. While it is clear that the reconstruction cost provides an unsupervised objective to harness the unlabeled examples, it is not clear how important the penalization is for each layer and what role it plays for fully-supervised tasks.

A second important change is the addition of Gaussian noise to the input and the hidden representations. While adding noise to the first layer is a part of denoising

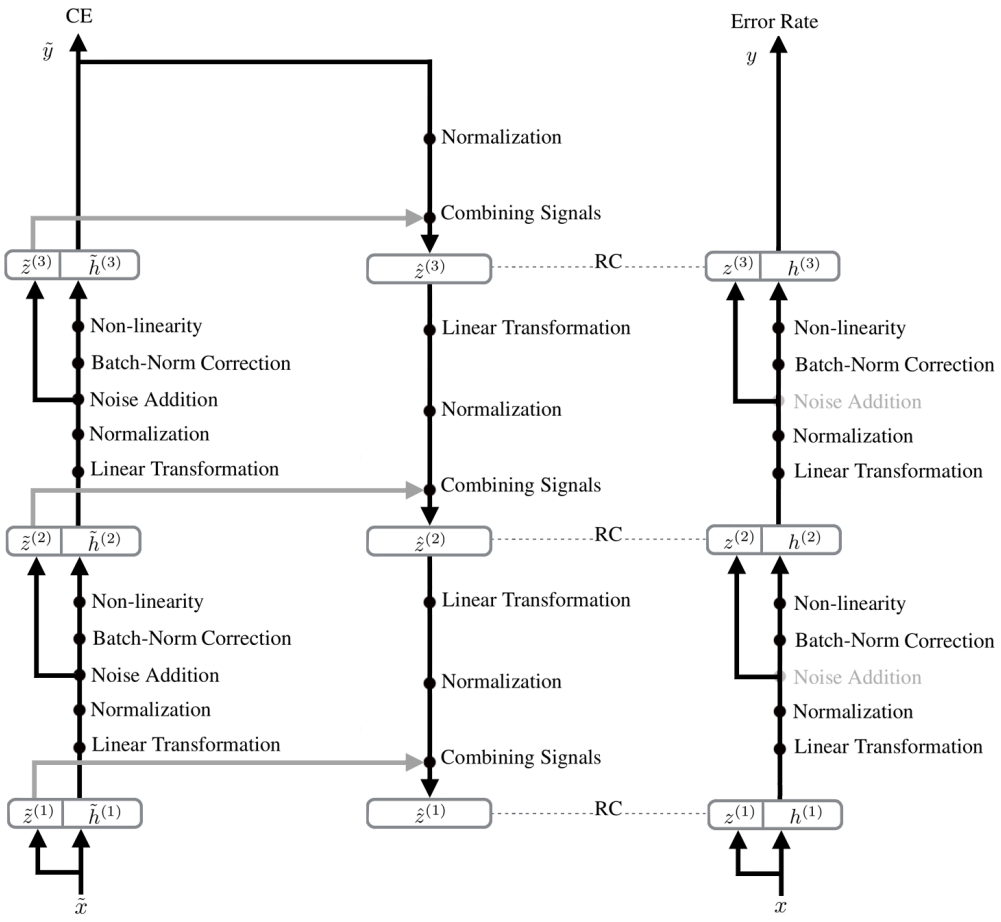

Figure 4.1 – The Ladder Network consists of two encoders (on each side of the figure) and one decoder (in the middle). At each layer of both encoders (equations 4.5 to 4.9), $z^{(l)}$ and $\tilde{z}^{(l)}$ are computed by applying a linear transformation and normalization on $h^{(l-1)}$ and $\tilde{h}^{(l-1)}$, respectively. The noisy version of the encoder (left) has an extra Gaussian noise injection term. Batch normalization correction (γ^l, β^l) and non-linearity are then applied to obtain $h^{(l)}$ and $\tilde{h}^{(l)}$. At each layer of the decoder, two streams of information, the lateral connection $\tilde{z}^{(l)}$ (gray lines) and the vertical connection $u^{(l+1)}$, are required to reconstruct $\hat{z}^{(l)}$ (equations 4.12 to 4.16). Acronyms CE and RC stand for Cross Entropy and Reconstruction Cost respectively. The final objective function is a weighted sum of all Reconstruction costs and the Cross Entropy cost.

autoencoder training, it is again not clear whether it is necessary to add this noise at every layer or not. We would also like to know if the noise helps by making the reconstruction task nontrivial and useful or just by regularizing the feed-forward network in a similar way noise-based regularizers like dropout (Srivastava et al., 2014) and adaptive weight noise (Graves, 2011).

Finally, the lateral skip connections are the most notable deviation from the

standard denoising autoencoder architecture. The way the vanilla Ladder Network combines the lateral stream of information $\tilde{z}^{(l)}$ and the downward stream of information $u^{(l+1)}$ is somewhat unorthodox. For this reason, we have conducted extensive experiments on both the importance of the lateral connections and the precise choice for the function that combines the lateral and downward information streams (which we refer to as the *combinator function*).

4.4 Experimental Setup

In this section, we introduce different variants of the Ladder Architecture and describe our experiment methodology. Some variants are derived by removal of one component of the model while other variants are derived by the replacement of that component with a new one. This enables us to isolate each component and observe its effects while other components remain unchanged. Table 4.1 depicts the hierarchy of the different variants and the baseline models.

4.4.1 Variants derived by removal of a component

Noise Variants

Different configurations of noise injection, penalizing reconstruction errors, and the lateral connection removal suggest four different variants:

— Add noise only to the first layer (FirstNoise).

— Only penalize the reconstruction at the first layer (FirstRecons), i.e. $\lambda^{(l \geq 1)}$ are set to 0.

— Apply both of the above changes: add noise and penalize the reconstruction only at the first layer (FirstN&R).

— Remove all lateral connections from FirstN&R. Therefore, equivalent to a denoising autoencoder with an additional supervised cost at the top, the encoder and the decoder are connected only through the topmost connection. We call this variant NoLateral.

Table 4.1 – Schematic ordering of the different models. All the variants of the VANILLA model are derived by either *removal* or *replacement* of a single component. The BASELINE is a multi-layer feedforward neural networks with the same number of layers and units as the vanilla model.

- Models
  - VANILLA
    - Removal of a component
      - Noise variants
        - FIRSTNOISE
        - FIRSTRECONS
        - FIRSTN&R
        - NOLATERAL
      - Vanilla Combinator variants
        - RANDINIT
        - REVINIT
        - NOSIG
        - NOMUL
        - LINEAR
    - Replacement of a component
      - MLP Combinators
        - MLP
        - AUGMENTED-MLP
      - Gaussian Combinators
        - GAUSSIAN
        - GATEDGAUSS
  - Baseline models
    - Feedforward network
    - Feedforward network + noise

Vanilla combinator variants

We try different variants of the vanilla combinator function that combines the two streams of information from the lateral and the vertical connections in an unusual way. As defined in equation 4.17, the output of the vanilla combinator depends on u, $\tilde{z}$, and $u \odot \tilde{z}$ [2], which are connected to the output via two paths, one linear and the other through a sigmoid non-linearity unit.

Note that the vanilla combinator is initialized in a very specific way (equation 4.19), which sets the initial weights for lateral connection $\tilde{z}$ to 1 and vertical connection u to 0. This particular scheme encourages the Ladder decoder path to learn

2. For simplicity, subscript i and superscript l are implicit from now on.

more from the lateral information stream $\tilde{z}$ than the vertical u at the beginning of training.

We explore two variants of the initialization scheme:

— Random initialization (RandInit): all per-element parameters are randomly initialized to $\mathcal{N}(0, 0.2)$.

— Reverse initialization (RevInit): all per-element parameters $w_{\{0,1\}z}$, $w_{\{0,1\}zu}$, and $b_{\{0,1\}}$ are initialized to zero while $w_{\{0,1\}u}$ and w_σ are initialized to one.

The simplest way of combining lateral connections with vertical connections is to simply add them in an element-wise fashion, similar to the nature of skip-connections in a recently published work on Residual Learning (He et al., 2015). We call this variant Linear combinator function. We also derive two more variants NoSig and NoMult in the way of stripping down the vanilla combinator function to the simple Linear one:

— Remove sigmoid non-linearity (NoSig). The corresponding per-element weights are initialized in the same way as the vanilla combinator.

— Remove the multiplicative term $\tilde{z} \odot u$ (NoMult).

— Simple linear combination (Linear)

$$g(\tilde{z}, u) = b + w_u \odot u + w_z \odot \tilde{z} \tag{4.21}$$

where the initialization scheme resembles the vanilla one in which w_z is initialized to one while w_u and b are initialized to zero.

4.4.2 Variants derived by replacement of a component

Gaussian combinator variants

Another choice for the combinator function with a probabilistic interpretation is the Gaussian combinator proposed in the original paper about the Ladder Architecture (Rasmus et al., 2015). Based on the theory in the Section 4.1 of (Valpola, 2014), assuming that both additive noise and the conditional distribution $P(z^{(l)}|u^{(l+1)})$ are Gaussian distributions, the denoising function is linear with respect to $\tilde{z}^{(l)}$. Hence, the denoising function could be a weighted sum over $\tilde{z}^{(l)}$ and a prior on $z^{(l)}$. The weights and the prior are modeled as a function of the vertical

signal:

$$g(\tilde{z}, u) = \nu(u) \odot \tilde{z} + (1 - \nu(u)) \odot \mu(u), \tag{4.22}$$

in which

$$\mu(u) = w_1 \odot \text{Sigmoid}(w_2 \odot u + w_3) + w_4 \odot u + w_5, \tag{4.23}$$

$$\nu(u) = w_6 \odot \text{Sigmoid}(w_7 \odot u + w_8) + w_9 \odot u + w_{10}. \tag{4.24}$$

Strictly speaking, $\nu(u)$ is not a proper weight, because it is not guaranteed to be positive all the time. To make the Gaussian interpretation rigorous, we explore a variant that we call GatedGauss, where equations 4.22 and 4.23 stay the same but 4.24 is replaced by:

$$\nu(u) = \text{Sigmoid}(w_6 \odot u + w_7). \tag{4.25}$$

GatedGauss guarantees that $0 < \nu(u) < 1$. We expect that $\nu(u)_i$ will be close to 1 if the information from the lateral connection for unit i is more helpful to reconstruction, and close to 0 if the vertical connection becomes more useful. The GatedGauss combinator is similar in nature to the gating mechanisms in other models such as Gated Recurrent Unit (Cho et al., 2014) and highway networks (Srivastava et al., 2015).

MLP (Multilayer Perceptron) combinator variants

We also propose another type of element-wise combinator functions based on fully-connected MLPs. We have explored two classes in this family. The first one, denoted simply as MLP, maps two scalars $[u, \tilde{z}]$ to a single output $g(\tilde{z}, u)$. We empirically determine the choice of activation function for the hidden layers. Preliminary experiments show that the Leaky Rectifier Linear Unit (LReLU) (Maas et al., 2013) performs better than either the conventional ReLU or the sigmoid unit. Our LReLU function is formulated as

$$\text{LReLU}(x) = \begin{cases} x, & \text{if } x \geq 0, \\ 0.1\,x, & \text{otherwise} \end{cases}. \tag{4.26}$$

We experiment with different numbers of layers and hidden units per layer in the MLP. We present results for three specific configurations: [4] for a single hidden layer of 4 units, [2, 2] for 2 hidden layers each with 2 units, and [2, 2, 2] for 3 hidden layers. For example, in the [2, 2, 2] configuration, the MLP combinator function is defined as:

$$g(\tilde{z}, u) = W_3\sigma\Big(W_2\sigma(W_1[u, \tilde{z}] + b_1) + b_2\Big) + b_3 \tag{4.27}$$

where W_1, W_2, and W_3 are 2×2 weight matrices; b_1, b_2, and b_3 are 2×1 bias vectors. The function $\sigma(.)$ is the Leaky ReLU activation function.

The second class, which we denote as AMLP (Augmented MLP), has a multiplicative term as an augmented input unit. We expect that this multiplication term allows the vertical signal ($u^{(l+1)}$) to override the lateral signal ($\tilde{z}$), and also allows the lateral signal to select where the vertical signal is to be instantiated. Since the approximation of multiplication is not easy for a single-layer MLP, we explicitly add the multiplication term as an extra input to the combinator function. AMLP maps three scalars $[u, \tilde{z}, u \odot \tilde{z}]$ to a single output. We use the same LReLU unit for AMLP.

We do similar experiments as in the MLP case and include results for [4], [2, 2] and [2, 2, 2] hidden layer configurations.

Both MLP and AMLP weight parameters are randomly initialized to $\mathcal{N}(0, \eta)$. η is considered to be a hyperparameter and tuned on the validation set. Precise values for the best η values are listed in Appendix.

4.4.3 Methodology

The experimental setup includes two semi-supervised classification tasks with 100 and 1000 labeled examples and a fully-supervised classification task with 60000 labeled examples for Permutation-Invariant MNIST handwritten digit classification. Labeled examples are chosen randomly but the number of examples for different classes is balanced. The test set is not used during all the hyperparameter search and tuning. Each experiment is repeated 10 times with 10 different but fixed random seeds to measure the standard error of the results for different parameter initializations and different selections of labeled examples.

All variants and the vanilla Ladder Network itself are trained using the ADAM

optimization algorithm (Kingma & Ba, 2014) with a learning rate of 0.002 for 100 iterations followed by 50 iterations with a learning rate decaying linearly to 0. Hyperparameters including the standard deviation of the noise injection and the denoising weights at each layer are tuned separately for each variant and each experiment setting (100-, 1000-, and fully-labeled). Hyperparmeters are optimized by either a random search (Bergstra & Bengio, 2012), or a grid search, depending on the number of hyperparameters involved (see Appendix for precise configurations).

4.5 Results & Discussion

Table 4.2 collects all results for the variants and the baselines. The results are organized into two main categories for all of the three tasks. Boxplots of four interesting variants are also shown in Figure 4.2. The Baseline model is a simple feed-forward neural network with no reconstruction penalty and Baseline+noise is the same network but with additive noise at each layer. The best results in terms of average error rate on the test set are achieved by the proposed AMLP combinator function: in the fully-supervised setting, the best average error rate is 0.569 ± 0.010, while in the semi-supervised settings with 100 and 1000 labeled examples, the averages are 1.002 ± 0.037 and 0.974 ± 0.021 respectively.

4.5.1 Variants derived by removal

The results in the table indicate that in the fully-supervised setting, adding noise either to the first layer only or to all layers leads to a lower error rate with respect to the baselines. Our intuition is that the effect of additive noise to layers is very similar to the weight noise regularization method (Graves, 2011) and dropout (Hinton et al., 2012).

In addition, it seems that removing the lateral connections hurts much more than the absence of noise injection or reconstruction penalty in the intermediate layers. It is also worth mentioning that hyperparameter tuning yields zero weights for penalizing the reconstruction errors in all layers except the input layer in the fully-supervised task for the vanilla model. Something similar happens for No-Lateral as well, where hyperparameter tuning yields zero reconstruction weights

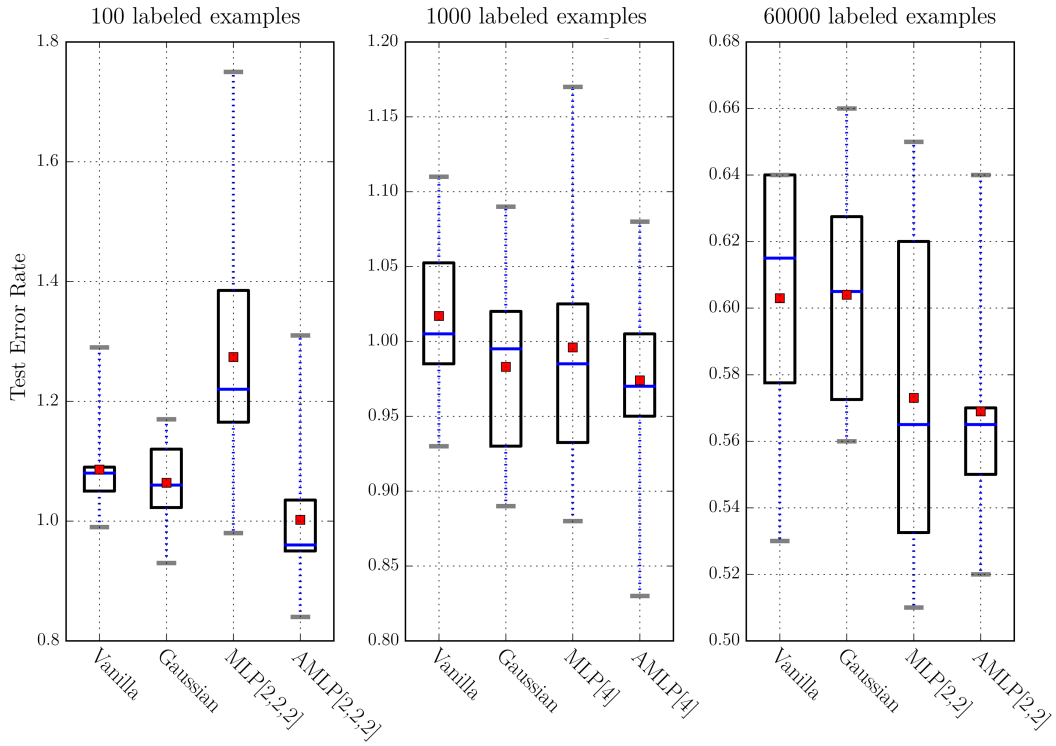

Figure 4.2 – Boxplots summarizing all individual experiments of four variants for 10 different seeds. Box boundaries represent the 25^{th} and 75^{th} percentiles. The blue line and the red square represent the median and the mean, respectively. The gray caps also show the minimum and maximum values. The variants that perform much worse than the vanilla Ladder Network are not plotted.

Table 4.2 – PI MNIST classification results for the vanilla Ladder Network and its variants trained on 100, 1000, and 60000 (full) labeled examples. AER and SE stand for Average Error Rate and its Standard Error of each variant over 10 different runs. BASELINE is a multi-layer feed-forward neural network with no reconstruction penalty.

	100		**1000**		**60000**	
Variant	**AER (%)**	**SE**	**AER (%)**	**SE**	**AER (%)**	**SE**
Baseline	25.804	± 0.40	8.734	± 0.058	1.182	± 0.010
Baseline+noise	23.034	± 0.48	6.113	± 0.105	0.820	± 0.009
Vanilla	1.086	± 0.023	1.017	± 0.017	0.608	± 0.013
FirstNoise	1.856	± 0.193	1.381	± 0.029	0.732	± 0.015
FirstRecons	1.691	± 0.175	1.053	± 0.021	0.608	± 0.013
FirstN&R	1.856	± 0.193	1.058	± 0.175	0.732	± 0.016
NoLateral	16.390	± 0.583	5.251	± 0.099	0.820	± 0.009
RandInit	1.232	± 0.033	1.011	± 0.025	0.614	± 0.015
RevInit	1.305	± 0.129	1.031	± 0.017	0.631	± 0.018
NoSig	1.608	± 0.124	1.223	± 0.014	0.633	± 0.010
NoMult	3.041	± 0.914	1.735	± 0.030	0.674	± 0.018
Linear	5.027	± 0.923	2.769	± 0.024	0.849	± 0.014
Gaussian	1.064	± 0.021	0.983	± 0.019	0.604	± 0.010
GatedGauss	1.308	± 0.038	1.094	± 0.016	0.632	± 0.011
MLP [4]	1.374	± 0.186	0.996	± 0.028	0.605	± 0.012
MLP [2, 2]	1.209	± 0.116	1.059	± 0.023	0.573	± 0.016
MLP [2, 2, 2]	1.274	± 0.067	1.095	± 0.053	0.602	± 0.010
AMLP [4]	1.072	± 0.015	**0.974**	± 0.021	0.598	± 0.014
AMLP [2, 2]	1.193	± 0.039	1.029	± 0.023	**0.569**	± 0.010
AMLP [2, 2, 2]	**1.002**	± 0.038	0.979	± 0.025	0.578	± 0.013

for all layers including the input layer. In other words, NOLATERAL and BASELINE+NOISE become the same models for the fully-supervised task. Moreover, the weights for the reconstruction penalty of the hidden layers are relatively smal in the semi-supervised task. This is in line with similar observations (relatively small weights for the unsupervised part of the objective) for the hybrid discriminant RBM (Larochelle & Bengio, 2008).

The third part of Table 4.2 shows the relative performance of different combinator functions by removal. Unsurprisingly, the performance deteriorates considerably if we remove the sigmoid non-linearity (NOSIG) or the multiplicative term

(NoMult) or both (Linear) from the vanilla combinator. Judging from the size of the increase in average error rates, the multiplicative term is more important than the sigmoid unit.

As described in Section 4.4.1 and Equation 4.19, the per-element weights of the lateral connections are initialized to ones while those of the vertical are initialized to zeros. Interestingly, the results are slightly worse for the RandInit variant, in which these weights are initialized randomly. The RevInit variant is even worse than the random initialization scheme. We suspect that the reason is that the optimization algorithm finds it easier to reconstruct a representation z starting from its noisy version $\tilde{z}$, rather than starting from an initially arbitrary reconstruction from the untrained upper layers. Another justification is that the initialization scheme in Equation 4.19 corresponds to optimizing the Ladder Network as if it behaves like a stack of decoupled DAEs initially, therefore during early training it is like that the Auto-Encoders are trained more independently.

4.5.2 Variants derived by replacements

The Gaussian combinator performs better than the vanilla combinator. GatedGauss, the other variant with strict $0 < \sigma(u) < 1$, does not perform as well as the one with unconstrained $\sigma(u)$. In the Gaussian formulation, $\tilde{z}$ is regulated by two functions of u: $\mu(u)$ and $\sigma(u)$. This combinator interpolates between the noisy activations and the predicted reconstruction, and the scaling parameter can be interpreted as a measure of the certainty of the network.

Finally, the AMLP model yields state-of-the-art results in all of 100-, 1000- and 60000-labeled experiments for PI MNIST. It outperforms both the MLP and the vanilla model. The additional multiplicative input unit $\tilde{z} \odot u$ helps the learning process significantly.

4.5.3 Probabilistic Interpretations of the Ladder Network

Since many of the motivations behind regularized autoencoder architectures are based on observations about generative models, we briefly discuss how the Ladder Network can be related to some other models with varying degrees of probabilistic interpretability. Considering that the components that are most defining of the Ladder Network seem to be the most important ones for semi-supervised learning

in particular, comparisons with generative models are at least intuitively appealing to get more insight about how the model learns about unlabeled examples.

By training the individual denoising autoencoders that make up the Ladder Network with a single objective function, this coupling goes as far as encouraging the lower levels to produce representations that are going to be easy to reconstruct by the upper levels. We find a similar term (-log of the top-level prior evaluated at the output of the encoder) in hierarchical extensions of the variational autoencoder (Rezende et al., 2014; Bengio, 2014). While the Ladder Network differs too much from an actual variational autoencoder to be treated as such, the similarities can still give one intuitions about the role of the noise and the interactions between the layers. Conversely, one also may wonder how a variational autoencoder might benefit from some of the components of Ladder Networks like Batch Normalization and multiplicative connections.

When one simply views the Ladder Network as a peculiar type of denoising autoencoder, one could extend the recent work on the generative interpretation of denoising autoencoders (Alain & Bengio, 2013; Bengio et al., 2013b) to interpret the Ladder Network as a generative model as well. It would be interesting to see if the Ladder Network architecture can be used to generate samples and if the architecture's success at semi-supervised learning translates to this profoundly different use of the model.

5 Conclusion

The thesis systematically compares different variants of the recent Ladder Network architecture (Rasmus et al., 2015; Valpola, 2014) with two feedforward neural networks as the baselines and the standard architecture (proposed in the original paper). Comparisons are done in a deconstructive way, starting from the standard architecture. Based on the comparisons of different variants we conclude that:

- Unsurprisingly, the reconstruction cost is crucial to obtain the desired regularization from unlabeled data.
- Applying additive noise to each layer and especially the first layer has a regularization effect which helps generalization. This seems to be one of the most important contributors to the performance on the fully supervised task.
- The lateral connection is a vital component in the Ladder architecture to the extent that removing it considerably deteriorates the performance for all of the semi-supervised tasks.
- The precise choice of the combinator function has a less dramatic impact, although the vanilla combinator can be replaced by the Augmented MLP to yield better performance, allowing us to improve the state-of-the-art on Permutation-Invariant MNIST for semi- and fully-supervised settings.

We hope that these comparisons between different architectural choices will help to improve understanding of semi-supervised learning's success for the Ladder Network and like architectures, and perhaps even deep architectures in general.

A Hyperparameter Selection

Here we provide the best hyperparameter combinations we have found for different variants in different settings. We consider the standard deviation of additive Gaussian noise and the reconstruction penalty weights in the decoder as the hyperparameters. For each variant, we fix the best hyperparameters tuned on the validation set and run the variant 10 times with 10 different but fixed data seeds (used to choose 100 or 1000 labeled examples).

Depending on each variant and its hyperparameter space, we used either random search or grid search. Table A.1 specifies the search space for hyperparameters and tables A.2, A.3, and A.4 collect the best hyperparameter combinations for each experiment setting. In the case of MLP and AMLP combinator functions, standard deviation of the Gaussian initialization η is chosen from a grid of $(0.0001, 0.006, 0.0125, 0.025, 0.05)$. The best η values are listed in Table A.5.

Table A.1 – Two different hyperparameter search methods. For random search, we run 20 random hyperparameter combinations for each variant and in each task.

Search method	Noise stddev ($\times 10^{-1}$)	Reconstruction weights search space
Random search	(0, 0, 0, 0, 0, 0, 0) (1, 1, 1, 1, 1, 1, 1) (2, 2, 2, 2, 2, 2, 2) (3, 3, 3, 3, 3, 3, 3) (4, 4, 4, 4, 4, 4, 4) (5, 5, 5, 5, 5, 5, 5) (6, 6, 6, 6, 6, 6, 6) (7, 7, 7, 7, 7, 7, 7) (8, 8, 8, 8, 8, 8, 8)	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (50.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (100.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (500.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (800.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (1000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (2000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (4000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (6000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (500, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1) (1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1) (2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
Grid 100 & 1000	(2, 2, 2, 2, 2, 2, 2) (3, 3, 3, 3, 3, 3, 3) (4, 4, 4, 4, 4, 4, 4)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1) (2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2) (5000, 50.0, 0.5, 0.5, 0.5, 0.5, 0.5) (10000, 100.0, 1.0, 1.0, 1.0, 1.0, 1.0)
Grid 60000	(2, 2, 2, 2, 2, 2, 2) (3, 3, 3, 3, 3, 3, 3) (4, 4, 4, 4, 4, 4, 4)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (1000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (2500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) (5000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

Table A.2 – Best hyperparameters for the semi-supervised task with 100 labeled examples.

Variant	Search method	Best noise stddev ($\times 10^{-1}$)	Best reconstruction weights
Baseline+noise	Random	(3, 3, 3, 3, 3, 3, 3)	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
Vanilla	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
FirstNoise	Random	(6, 0, 0, 0, 0, 0, 0)	(1000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
FirstRecons	Random	(3, 3, 3, 3, 3, 3, 3)	(1000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
FirstN&R	Random	(6, 0, 0, 0, 0, 0, 0)	(1000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
NoLateral	Random	(7, 0, 0, 0, 0, 0, 0)	(100.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
RandInit	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
RevInit	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
NoSig	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
NoMult	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
Linear	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
Gaussian	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
GatedGauss	Grid	(2, 2, 2, 2, 2, 2, 2)	(2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
MLP[4]	Grid	(2, 2, 2, 2, 2, 2, 2)	(5000, 50.0, 0.5, 0.5, 0.5, 0.5, 0.5)
MLP[2,2]	Grid	(2, 2, 2, 2, 2, 2, 2)	(2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
MLP[2,2,2]	Grid	(2, 2, 2, 2, 2, 2, 2)	(10000, 100.0, 1.0, 1.0, 1.0, 1.0, 1.0)
AMLP[4]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
AMLP[2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
AMLP[2,2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.2, 0.2, 0.2, 0.2, 0.2)

Table A.3 – Best hyperparameters for the semi-supervised task with 1000 labeled examples.

Variant	Search method	Best noise stddev ($\times 10^{-1}$)	Best reconstruction weights
Baseline+noise	Random	(2, 2, 2, 2, 2, 2, 2)	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
Vanilla	Grid	(2, 2, 2, 2, 2, 2, 2)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
FirstNoise	Random	(6, 0, 0, 0, 0, 0, 0)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
FirstRecons	Random	(3, 3, 3, 3, 3, 3, 3)	(4000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
FirstN&R	Random	(6, 0, 0, 0, 0, 0, 0)	(1000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
NoLateral	Random	(6, 0, 0, 0, 0, 0, 0)	(100.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
RandInit	Grid	(2, 2, 2, 2, 2, 2, 2)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
RevInit	Grid	(2, 2, 2, 2, 2, 2, 2)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
NoSig	Grid	(2, 2, 2, 2, 2, 2, 2)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
NoMult	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
Linear	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
Gaussian	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
GatedGauss	Grid	(2, 2, 2, 2, 2, 2, 2)	(1000, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
MLP[4]	Grid	(3, 3, 3, 3, 3, 3, 3)	(10000, 100.0, 1.0, 1.0, 1.0, 1.0, 1.0)
MLP[2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(5000, 50.0, 0.5, 0.5, 0.5, 0.5, 0.5)
MLP[2,2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
AMLP[4]	Grid	(3, 3, 3, 3, 3, 3, 3)	(5000, 50.0, 0.5, 0.5, 0.5, 0.5, 0.5)
AMLP[2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 20.0, 0.2, 0.2, 0.2, 0.2, 0.2)
AMLP[2,2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 10.0, 0.2, 0.2, 0.2, 0.2, 0.2)

Table A.4 – Best found hyperparameters for the task of semi-supervised with 60000 labeled examples.

Variant	Search method	Best noise stddev ($\times 10^{-1}$)	Best reconstruction weights
Baseline+noise	Random	(3, 3, 3, 3, 3, 3, 3)	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
Vanilla	Grid	(3, 3, 3, 3, 3, 3, 3)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
FirstNoise	Random	(6, 0, 0, 0, 0, 0, 0)	(500, 10.0, 0.1, 0.1, 0.1, 0.1, 0.1)
FirstRecons	Random	(3, 3, 3, 3, 3, 3, 3)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
FirstN&R	Random	(6, 0, 0, 0, 0, 0, 0)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
NoLateral	Random	(6, 0, 0, 0, 0, 0, 0)	(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
RandInit	Grid	(3, 3, 3, 3, 3, 3, 3)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
RevInit	Grid	(3, 3, 3, 3, 3, 3, 3)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
NoSig	Grid	(3, 3, 3, 3, 3, 3, 3)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
NoMult	Grid	(3, 3, 3, 3, 3, 3, 3)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
Linear	Grid	(3, 3, 3, 3, 3, 3, 3)	(500, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
Gaussian	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
GatedGauss	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
MLP[4]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
MLP[2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
MLP[2,2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(1000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
AMLP[4]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
AMLP[2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
AMLP[2,2,2]	Grid	(3, 3, 3, 3, 3, 3, 3)	(2000, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

Table A.5 – Best MLP initialization η for all settings.

MLP variant	100 labels	1000 labels	fully-labeled
MLP[4]	0.006	0.006	0.0125
MLP[2,2]	0.05	0.0125	0.05
MLP[2,2,2]	0.025	0.025	0.05
AMLP[4]	0.006	0.025	0.0125
AMLP[2,2]	0.0125	0.0125	0.025
AMLP[2,2,2]	0.006	0.006	0.006

Bibliography

Guillaume Alain and Yoshua Bengio. What regularized auto-encoders learn from the data generating distribution. *International Conference on Learning Representations*, 2013.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2009.

Yoshua Bengio. How auto-encoders could provide credit assignment in deep networks via target propagation. Technical report, arXiv arXiv:1407.7906, 2014.

Yoshua Bengio, Aaron Courville, and Pierre Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions*, 2013a.

Yoshua Bengio, Li Yao, Guillaume Alain, and Pascal Vincent. Generalized denoising auto-encoders as generative models. *Neural Information Processing Systems*, 2013b.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 2012.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua

Bengio. Theano: a cpu and gpu math expression compiler. *Proceedings of the Python for scientific computing conference (SciPy)*, 2010.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

Aaron C Courville, James Bergstra, and Yoshua Bengio. A spike and slab restricted boltzmann machine. *International Conference on Artificial Intelligence and Statistics*, pp. 233–241, 2011.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Domain adaptation for large-scale sentiment classification: A deep learning approach. *Proceedings of the 28th International Conference on Machine Learning*, pp. 513–520, 2011.

Alex Graves. Practical variational inference for neural networks. *Advances in Neural Information Processing Systems*, pp. 2348–2356, 2011.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006.

Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 2006.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. Technical report, arXiv arXiv:1207.0580, 2012.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 1989.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

Diederik P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. *Advances in Neural Information Processing Systems*, pp. 3581–3589, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted boltzmann machines. *Proceedings of the 25th international conference on Machine learning*, 2008.

Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 609–616, 2009.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. *Proceedings of the 30th international conference on Machine learning*, 2013.

Razvan Pascanu, Yann N Dauphin, Surya Ganguli, and Yoshua Bengio. On the saddle point problem for non-convex optimization. *arXiv preprint arXiv:1405.4604*, 2014.

Marc'Aurelio Ranzato and Martin Szummer. Semi-supervised learning of compact document representations with deep networks. *Proceedings of the 25th international conference on Machine learning*, 2008.

Antti Rasmus, Harri Valpola, Mikko Honkala, Mathias Berglund, and Tapani Raiko. Semi-supervised learning with ladder network. *arXiv preprint arXiv:1507.02672*, 2015.

Danilo J. Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. *International Conference of Machine Learning*, 2014.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 2014.

Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. *Advances in Neural Information Processing Systems*, pp. 2368–2376, 2015.

Simon J Thorpe and Michèle Fabre-Thorpe. Seeking categories in the brain. *Science*, 2001.

Harri Valpola. From neural pca to deep unsupervised learning. *arXiv preprint arXiv:1411.7783*, 2014.

Bart van Merriënboer, Dzmitry Bahdanau, Vincent Dumoulin, Dmitriy Serdyuk, David Warde-Farley, Jan Chorowski, and Yoshua Bengio. Blocks and fuel: Frameworks for deep learning. *arXiv preprint arXiv:1506.00619*, 2015.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. *Proceedings of the 25th international conference on Machine learning*, pp. 1096–1103, 2008.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 2010a.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 2010b.