Literature Survey - (Title pending)

Ash

March 14, 2018

Contents

List of Figures

Chapter 1

Introduction

Very brief background Deep learning good for large data set. Motivation is to work with less training examples

Existing work on meta learning and continuous learning

Gap the literature

Describe the Problem A system that can do continual learning. Refer to this doc: https://paper.dropbox.com/doc/A
classes-an-existing-classifier-RdKxXHh7M9OWbHvEvCCsV We want it to be scalable with respect to the number of classes So it should work faster than nearest neighbour based approaches for large number of classes

High level how you will solve it and why it is different from existing work

Brief description of experimental setup

Chapter 2

Background

(Week 2)

2.1 Hand Engineered and Learnt Features

(Week 2)
Features are a general term for characteristic attributes which exist across all samples in a data-set or domain. These features were traditionally hand-engineered by machine learning experts, carefully selecting the base-components of which the data-set in question appears to comprise.

A fundamental problem with hand-engineered features is that it imposes human knowledge onto a problem to be solved by a computer. Furthermore, key features for complex data such as images and video are incredibly difficult to ascertain – especially if desiring generic, transferable features. With the rise of neural networks – specifically CNNs, which will be discussed in detail later – feature-learning has become the norm. This essentially takes the task of feature engineering and solves it in a data-driven manner.

2.2 Supervised Learning

(Week 2) Supervised learning is a machine learning strategy whereby the target solution is presented after each training iteration. This differs from unsupervised learning in that unsupervised learning has no direct target to learn from and is used to find underlying commonalities or patterns in data. Supervised learning is the most commonly used method for image and video tasks, as typically the objective is to perform tasks where the target is well-defined. Common supervised learning tasks are *image classification* - where the objective is to assign an input image a label from a fixed set of categories; *localisation* - where the objective is to produce the coordinates of an object of interest from the input image; and *detection* - which combines the previous two tasks.

There are a multitude of supervised and unsupervised learning problems, but as with almost all meta-learning strategies I will focus primarily on the supervised task of image classification using neural networks.

2.3 Optimisation

(Week 2)

2.3.1 Loss

The process of optimising a machine learning system is to present it with a target of some sort – either in the supervised or unsupervised setting – and compute a numerical quantity called *loss* or *cost*. The loss is a scalar value which is representative of how "badly" the system has performed inference given the input image. It is the system's objective to minimise this value through some optimisation algorithm. The loss function is specifically chosen for the task at hand, *cross-entropy* being a common choice for image classification tasks and *mean-squared (L2) error* for localisation.

Symbols

Before discussing loss functions, it's important to understand the inputs and outputs of a machine learning system when performing image classification.
For a system that makes predictions between N classes, its input is a vector of image features $\boldsymbol{x}$ - usually the raw pixel values. The system's output is a vector $\hat{\boldsymbol{y}}$ of length N, where each of the output values $\hat{\boldsymbol{y}}_i$ is a score for class i being the correct answer. The loss $\mathcal{L}$, as described above, is a function of the predictions $\hat{\boldsymbol{y}}$ and the target values $\boldsymbol{y}$.
The target values are typically encoded as a *one-hot* vector of length N, which is all zeros with a 1 in the position of the correct class.

Softmax

As the system's outputs aren't normalised and thus cannot be interpreted as a true confidence measure, the outputs normally go through a softmax function σ.

$$\sigma(\hat{\boldsymbol{y}})_i = \frac{e^{\hat{y}_i}}{\sum_{k=1}^{N} e^{\hat{y}_i}} \tag{2.1}$$

The softmax function (eq 2.1) squashes the arbitrary scores into a vector such that its values sum to 1 and are each in the range $[0, 1]$. The resultant values can be interpreted as the probability of the input image falling into each of the classes.

Cross-Entropy Loss

With the machine learning system producing a normalised probability distribution across classes, those values need be compared with the targets to produce a scalar loss value. Cross-entropy loss, otherwise known as *log loss*, penalises for differences between predicted values and targets, with the penalty growing harsher for further-away predictions as demonstrated in fig 2.1.
For a vector of predictions $\hat{\boldsymbol{y}}$ and a one-hot target vector $\boldsymbol{y}$, the cross-entropy loss is:

$$H(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{k=1}^{N} y_k log(\hat{y}_k) \tag{2.2}$$

For the special case of *binary* cross-entropy (as shown in fig 2.1) where number of classes N is 2, the network's outputs are generally reduced to a single scalar value and cross entropy calculated as:

$$H(\hat{y}, y) = -(ylog(\hat{y}) + (1 - y)log(1 - \hat{y}) \tag{2.3}$$

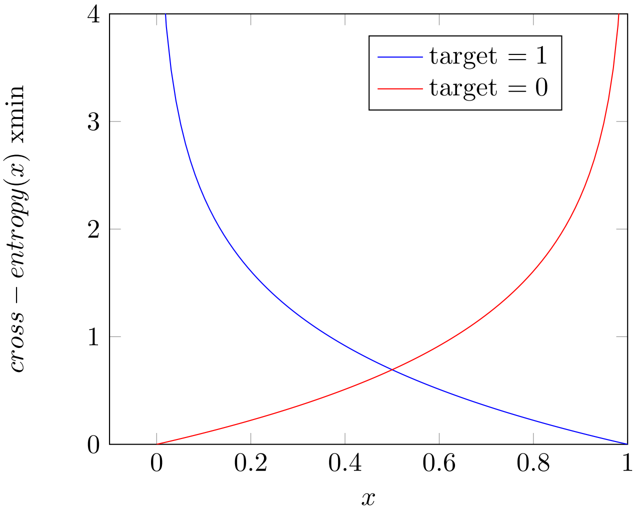

Figure 2.1: Binary Cross-Entropy

Mean-Squared Error (L2)

Mean-squared error is used for regression tasks, where the objective is to predict a quantitative value rather than a measure of probability. As L2 loss minimises the average error between the predictions $\hat{\boldsymbol{y}}$ and targets $\boldsymbol{y}$, the system learns to make predictions which lie in the mean position of these, which is generally ideal for regression tasks where there is one solution.

$$L2(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_{k=1}^{N} (y_k - \hat{y}_k)^2 \tag{2.4}$$

2.4 Neural Networks

Artifical Neural Networks (*ANNs*) are machine learning systems loosely inspired by the functioning of the biological neural networks of the brain. They are composed of artificial neurons which transmit signals from one another in the form of a non-linear function of the sum of the incoming inputs. ANNs model unknown functions of arbitrary complexity, with their representational power a function of their size.

If we look at the structure of the simplest neuron possible f_1 (see fig (make figure)) we see that it is composed of two components - a weight w_1 and a bias b_1. Passing a value x through a neuron f_1 is equivalent to computing the linear function $f_1(x) = w_1 x + b_1$. If the outputs of this neuron are then passed into a similar neuron f_2, we end up with the composite function

$$\begin{aligned}(f_2 \circ f_1)(x) &= (w_2(w_1 x + b_1) + b_2) & (2.5)\\ &= w_2 w_1 x + b_1 w_2 + b_2 & (2.6)\end{aligned}$$

which is still a linear function of x. This is true for any number of sequential neurons, meaning that any composition of linear neurons is only as good as a single neuron. Having the capacity to produce linear relationships is only useful if the function being modelled is, itself, a linear function. For more complex modelling tasks – which are encountered more often than not – non-linearities need to be introduced into the network. In making a neuron a non-linear function, the problem with composite functions noted above no-longer exists; adding additional neurons increases the representational power of the model.

2.4.1 Activation Functions / Non-Linearities

Activation functions (also known as non-linearities) are a critical component of neural networks as they add the capacity to model non-linear relationships. The inspiration for activation functions is drawn – once again – from the operation of biological neural networks, whereby neurons are only "activated" given sufficient input signal. In modern neural networks there are only a few activation functions regularly used:

- **Sigmoid** $= \frac{1}{1+e^{-x}}$: The Sigmoid activation function (also known as the *logistic function*) has the nice property that $(\forall x \in \mathbb{R}) Sigmoid(x) \in (0, 1)$ which is useful as a way of normalising values, especially when the output is to be interpreted as a probability.

- **TanH** $= \frac{e^{2x}-1}{e^{2x}+1}$: The hyperbolic tangent function is similar to sigmoid, but maps numbers to the range $(-1, 1)$.

- **ReLU** $= \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$ The Rectified Linear Unit maps numbers to the range $[0, \infty)$ and has the advantage that it is much more computationally efficient than the above two activation functions; in most cases it yields better results.

- **Leaky ReLU** $= \begin{cases} 0.01x, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$ The Leaky Rectified Linear Unit operates like ReLU, but allows numbers less than zero to "leak" through - this is helpful during *backpropogation*, which is discussed at length in section 2.4.4.

2.4.2 Fully-Connected Layers

The arrangement and structure of neurons discussed thus far hasn't been very practical, in that we were only considering a chain of continuous neurons one after another with only one input and output. Neurons will typically receive multiple inputs and produce a weighted sum of those inputs. A neuron given N inputs x_i with weights w_i and bias b would then form the linear equation $w_1 x_1 + w_2 x_2 + ... + w_N x_N + b$.

For a system consisting of multiple inputs, we want to allow diverse interactions between them. This is achieved by what's known as a *Fully-Connected Layer* of neurons, where each output from the previous layer is passed as input to each of the following layer's neurons. Networks are generally grouped into layers to provide a nice abstraction away from the hundreds or thousands of neurons inside, see fig (Create a nice picture of this).

2.4.3 Stochastic Gradient Descent (SGD)

(Week 2)
An ANN begins with randomly generated weights and biases $\boldsymbol{W}$ and $\boldsymbol{B}$, which are collectively referred to as the "parameters" or "weights" and indicated by $\boldsymbol{\theta}$. The objective of an ANN is to select weights $\boldsymbol{\theta}$ that minimise the error computed by the loss function $\mathcal{L}$ (section 2.3.1). Linear functions $f(\boldsymbol{x}, \boldsymbol{\theta})$ can be minimised by analytical techniques, but complex neural networks must be iteratively optimised by numerical methods.
(Put in the argmax/minimisation objective function)

With the loss between a prediction $\hat{y}$ and target y, we can compute the gradients of the parameters and make a small step in the direction which will reduce the loss for the given example (see fig (make loss surface figure)). When performing this operation over the entire data-set at once, this is known as gradient descent. This is usually not an option as the computational resources required for full gradient descent are prohibitive. If we instead repeat this operation for different examples until we have stabilised the loss to a low value, we have Stochastic Gradient Descent. The most common variant to this technique is known as Batch Gradient Descent, where instead of computing the loss and performing an update to the parameters on a per-example basis, the process is applied once per *batch* of examples. It facilitates more stable learning, as the loss doesn't fluctuate as much as between single examples. Batch Gradient Descent is the basis for most ANN optimisation, although we'll discuss modern variants in section 2.7.

2.4.4 Gradients and Backpropagation

(Week 2)
We shall consider a general ANN σ with 2 fully-connected layers f_1, f_2 parameterised by $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2\}$ which can be represented as:

$$\hat{\boldsymbol{y}} = \sigma(\boldsymbol{x}, \boldsymbol{\theta}) = f_2(f_1(\boldsymbol{x}, \boldsymbol{\theta}_1), \boldsymbol{\theta}_2) \tag{2.7}$$

That is, the input $\boldsymbol{x}$ is fed through layer f_1 then f_2. We will also consider an arbitrary loss function $\mathcal{L}$ which compares the predictions $\hat{\boldsymbol{y}}$ with targets $\boldsymbol{y}$ and produces a scalar loss value:

$$\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) \tag{2.8}$$

As the input $\boldsymbol{x}$ and target $\boldsymbol{y}$ is fixed, we may only change the parameters $\boldsymbol{\theta}$ to improve the loss. As explained in section 2.4.3, we wish to make incremental changes to our parameters where each change decreases our loss value. We do so by computing the gradient of the parameters with respect to the loss value:

$$\frac{\partial \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})}{\partial \boldsymbol{\theta}} \tag{2.9}$$

If we are to find the gradients of the loss function with respect to the final layer's weights, we will have the equation

$$\frac{\partial \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})}{\partial \boldsymbol{\theta}_2} \tag{2.10}$$

which is considered to be easily calculable. However, if we wish to find the gradient of weights further towards the start of the network, we cannot work those out directly and instead need to make use of the differentiation chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x} \tag{2.11}$$

To find the gradient of the loss function with respect to the first layer's weights, and considering that $\frac{\partial \mathcal{L}(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{\theta}_2}$ is calculable, we simply need apply the chain rule as such:

$$\frac{\partial \mathcal{L}(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{\theta}_1} = \frac{\partial \mathcal{L}(\hat{\boldsymbol{y}},\boldsymbol{y})}{\partial \boldsymbol{\theta}_2}\frac{\partial \boldsymbol{\theta}_2}{\partial \boldsymbol{\theta}_1} \tag{2.12}$$

That is, once we know the gradient of the loss with respect to the second layer's weights, we can compute the gradient of the second layer's weights with respect to the first layer's weights and multiply the two to find the gradient of the loss with respect to the first layer's weights. This generalises to neural networks with any number of layers of differing types. This is an intuitive relationship, as we are essentially calculating the compound contribution that a change in any one parameter's value will have on the final output – the loss.

This technique used since the 1970s [3] is aptly named *backpropagation of error gradient*, as it involves the propagation of the gradient of the error – or loss – from the end of the network back.

Returning to the concept of SGD (section 2.4.3) which was introduced on a conceptual level, we can now delve deeper into the application of the algorithm. Assuming that we have computed the gradient of each parameter in the network for the given examples in our batch, we can visualise this as a "loss landscape", whereby moving the parameters down-hill results in a reduction in the loss. Adjusting a parameter by the negative of the gradient will result in a "step" that moves the parameter closer to a local minimum, with the objective to reach the lowest point possible. A configurable hyperparameter is the *learning-rate*, commonly represented by α, which is the "size" of the step to take - that is, the coefficient of the negative of the gradient to apply.

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\Delta_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta},\hat{\boldsymbol{y}},\boldsymbol{y}) \tag{2.13}$$

For brevity we will from now write the gradient of the parameters $\boldsymbol{\theta}$ with respect to the loss function $\mathcal{L}$ as $\Delta_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$.

The problem that is quickly encountered is how large a value to set the learning rate α. Too small a step and it will take a long time to reach the minimum; too large and you may over-shoot it (see fig (make a figure)). The example in (fig (make a figure)) is a very low-dimensional loss landscape where the corect direction to move seems very logical, but higher dimensional spaces with a greater number of parameters result in complex landscapes with many local minima. We also want to slow down when nearing the bottom of a minimum so we can properly reach the lowest point. If we take this into consideration, and the fact that for any mini-batch the loss landscape will be different due to noise in small sample sizes, choosing a fixed learning rate becomes problematic. In practice, many people simply set up a learning rate schedule, where they decrease the learning rate at intervals, but it is hard to get right. It is with this in mind that adaptive optimizers came about.

2.4.5 Adaptive Optimizers

If we consider a data-set whereby each mini-batch has some noise but that there is an optimal parameterisation for the entire set, we could visualise the loss landscape as shifting slightly for each mini-batch, but where there is a common direction towards which most samples will lead. Standard SGD will just apply a fixed step-size for all updates which leads the parameters to oscillate along the greatest descent angle, regardless of past updates.

SGD with Momentum

It is with the idea of a consistent optimization direction that momentum comes into play. This is a technique of updating parameters while taking past updates into consideration to find the "common direction" in which to move. This is done by using an update vector which intuitively acts like the momentum of a ball rolling down a hill. For any update step, the direction vector is updated with a scaled contribution from the current gradient direction, and that vector is used to update the parameters. It essentially dampens oscillating movement, as the direction vector compounds contributions in the same direction as given in the following equation

$$\boldsymbol{v}_t = \gamma \boldsymbol{v}_{t-1} + \alpha \Delta_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \tag{2.14}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \boldsymbol{v}_t \tag{2.15}$$

where γ is the momentum term which indicates how much movement we wish to carry forward from previous time-steps.

Nesterov Accelerated Gradient Descent

If we consider momentum to compound the previous slopes such as a ball rolling down a hill, we end up with a problem where the momentum may cause the parameters to overshoot the minimum. Nesterov Accelerated Gradient Descent pre-emptively considers post-step parameters and makes adjustments to the step *actually* taken. It does so by approximating the position after a momentum parameter update ($\boldsymbol{\theta} - \gamma \boldsymbol{v}_{t-1}$), and computes the loss gradient not with respect to the current parameters, but to the approximate future position of the parameters. This optimisation strategy essentially glances into the future and makes a pre-emptive correction.

$$\boldsymbol{v}_t = \gamma \boldsymbol{v}_{t-1} + \alpha \Delta_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta} - \gamma \boldsymbol{v}_{t-1}) \tag{2.16}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \boldsymbol{v}_t \tag{2.17}$$

Adagrad

Adagrad is the first of the true "adaptive" optimizers, in that it adjusts the learning rate on a per-parameter basis, taking past gradients into consideration. The Adagrad update rule is

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\gamma}{\sqrt{g_{t,i} + \epsilon}} \mathcal{L}(\theta_{t,i}) \tag{2.18}$$

where θ_i is the ith parameter, γ is the learning rate, g_i is the square of the sum of the squares of the gradients with respect to θ_i up to step, ϵ is a small number to avoid dividing by zero, and $\mathcal{L}(\theta_i)$ is the gradient of the loss with respect to θ_i. This optimizer has the benefit that while rarely performing as well as SGD with well-chosen hyperparameters, the Adagrad's default learning rate of 0.01 consistently yields very good results.

Adadelta and RMSprop

The accumulation of squared gradients in the denominator of Adagrad means that that the learning rate continues shrinking and eventually becomes small enough to entirely stop training. Adadelta seeks to resolve this by instead of storing the accumulated sum of squared gradients, keeping a running average

$$E[g^2]_{t,i} = \gamma E[g^2]_{t-1,i} + (1 - \gamma) g_{t,i}^2 \tag{2.19}$$

and updating equation 2.18 to be

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\gamma}{\sqrt{E[g^2]_{t,i} + \epsilon}} \mathcal{L}(\theta_{t,i}) \tag{2.20}$$

RMSprop [1] is in effect identical to Adadelta, and were both developed to address the diminishing learning rate proplem of Adagrad.

Adam

Adaptive Moment Estimation (Adam) [2] builds from the previous adaptive optimizers by not only storing exponentially decaying averages of past squared gradients $v_{t,i}$, but by also keeping an exponentially decaying average of past gradients $m_{t,i}$ – note the lack of the word "squared".

$$m_{t,i} = \beta_1 m_{t-1,i} + (1 - \beta_1) g_{t,i} \tag{2.21}$$

$$v_{t,i} = \beta_2 v_{t-1,i} + (1 - \beta_2) g_{t,i}^2 \tag{2.22}$$

where β_1, β_2 are hyperparameters the proportion of past information carried forward, with the default values typically working well in practice. Replacing $E[\boldsymbol{g}^2]$ with $\boldsymbol{v}$, $\mathcal{L}(\boldsymbol{\theta})$ with $\boldsymbol{m}$, and a small change to the square-root we end up with

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\gamma}{\sqrt{v_{t,i}} + \epsilon} m_{t,i} \tag{2.23}$$

While not the most recent (I will omit a few other adaptive optimizers), Adam is rapidly becoming the most frequently used optimizer for its consistent convergence and ease-of-use.

2.5 Dealing with Small Training Data Sets

2.5.1 Overfitting

2.5.2 Transfer Learning

2.5.3 Few-Shot Learning

2.5.4 Meta Learning

Break into meta training, testing, episodes, etc.

2.6 Continuous Learning

Catastrophic forgetting

2.7 Modern Deep Learning Architectures

2.7.1 Convolutional Neural Networks

2.7.2 Recurrent Neural Networks

Chapter 3

Related Works

3.1 Meta Learning

Approaches in meta learning with neural networks are generally groupd into three categories.

3.1.1 Model Based

3.1.2 Metric Based

3.1.3 Optimization Based

3.2 Continuous Learning

Chapter 4

Proposal

Bibliography

[1] G. Hinton. *csc321 lecture 6*. 2015. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (visited on 03/14/2018).

[2] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[3] J. Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *Neural Networks* 61 (2015). Published online 2014; based on TR arXiv:1404.7828 [cs.NE], pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003.