Recap of Basic Neural Networks

(and some Deep Network Fundamentals)

Jonathon Hare and Antonia Marcu

Classical Types of Learning

- Supervised Learning - learn to predict an output when given an input vector
- Unsupervised Learning - discover a good internal representation of the input
- Reinforcement Learning - learn to select an action to maximize the expectation of future rewards (payoff)
- Semi-supervised Learning - learn with few labelled examples and many unlabelled ones

Other Types of Learning

- Self-supervised Learning - learn with targets induced by a prior on the unlabelled training data
- Active Learning - learn by seeking guidance from human or oracle when needed (iterative semi-supervised learning)
- Continual Learning - learn new tasks/classes sequentially (iterative supervised/unsupervised learning)
- Online learning - learning one example at a time sequentially (iterative supervised learning)

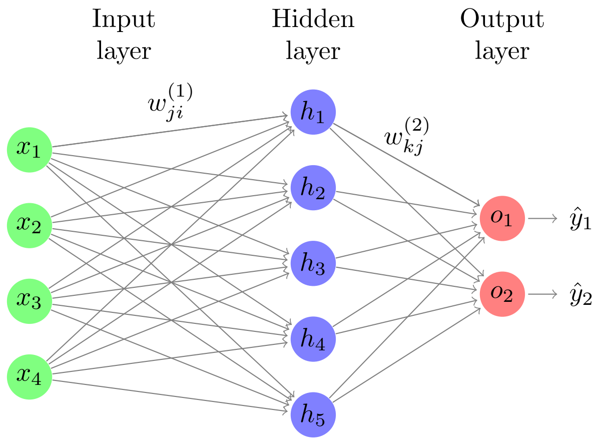
Two Types of Supervised Learning

- Regression: The machine is asked predict k numerical values given some input. The machine is a function $f : \mathbb{R}^n \to \mathbb{R}^k$.
- Classification: The machine is asked to specify which of k categories some input belongs to.
  - Multiclass classification - target is one of the k classes
  - Multilabel classification - target is some number of the k classes
  - In both cases, the machine is a function $f : \mathbb{R}^n \to \{1, ..., k\}$.
- It is most common for both types of algorithms to actually learn $\hat{f} : \mathbb{R}^n \to \mathbb{R}^k$.
- Note that there are lots of exceptions in the form the inputs (and outputs) can take though! We'll see lots of variations in the coming weeks.

How Supervised Learning Typically Works

- Start by choosing a model-class: $\hat{y} = f(\boldsymbol{x}; \boldsymbol{W})$ where the model-class f is a way of using some numerical parameters, $\boldsymbol{W}$, to map each input vector $\boldsymbol{x}$ to a predicted output $\hat{y}$.
- *Learning* means adjusting the parameters to reduce the discrepancy between the true target output y on each training case and the output $\hat{y}$, predicted by the model.

Let's look at a Multilayer Perceptron (without biases)...



Without loss of generality, we can write the above as:

$$\hat{\boldsymbol{y}} = g(f(\boldsymbol{x}; \boldsymbol{W}^{(1)}); \boldsymbol{W}^{(2)}) = g(\boldsymbol{W}^{(2)} f(\boldsymbol{W}^{(1)} \boldsymbol{x}))$$

where f and g are activation functions.

Common Activation Functions

- Identity
- Sigmoid (aka Logistic)
- Hyperbolic Tangent (tanh)
- Rectified Linear Unit (ReLU) (aka Threshold Linear)

Final layer activations

$$\hat{y} = g(\boldsymbol{W}^{(2)} f(\boldsymbol{W}^{(1)} \boldsymbol{x}))$$

- What form should the final layer function g take?
- It depends on the task (and on the chosen loss function)...
  - For regression it is typically linear (e.g. identity), but you might choose others if you say wanted to clamp the range of the network.
  - For binary classification (MLP has a single output), one would choose Sigmoid
  - For multilabel classification, typically one would choose Sigmoid
  - For multiclass classification, typically you would use the Softmax function

Softmax

The softmax is an activation function used at the output layer of a neural network that forces the outputs to sum to 1 so that they can represent a probability distribution across a discrete mutually exclusive alternatives.

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad \forall i = 1, 2, \ldots, K$$

- Note that unlike the other activation functions you've seen, softmax makes reference to all the elements in the output.
- The output of a softmax layer is a set of positive numbers which sum up to 1 and can be thought of as a probability distribution.
- Note:

$$\begin{aligned}
\frac{\partial\, \text{softmax}(\boldsymbol{z})_i}{\partial z_i} &= \text{softmax}(z_i)(1 - \text{softmax}(z_i)) \\
\frac{\partial\, \text{softmax}(\boldsymbol{z})_i}{\partial z_j} &= \text{softmax}(z_i)(1(i = j) - \text{softmax}(z_j)) \\
&= \text{softmax}(z_i)(\delta_{ij} - \text{softmax}(z_j))
\end{aligned}$$

Ok, so let's talk loss functions

- The choice of loss function depends on the task (e.g. classification/regression/something else)
- The choice also depends on the activation function of the last layer
  - For numerical reasons (see Log-Sum-Exp in a few slides) many times the activation is computed directly within the loss rather than being part of the model
  - Some classification losses require *raw outputs* (e.g. a linear layer) of the network as their input
    - These are often called *unnormalised log probabilities* or *logits*
    - An example would be hinge-loss used to create a Support Vector Machine that maximises the margin — e.g.: $\ell_{hinge}(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$ with a true label, $y \in \{-1, 1\}$, for binary classification.
- There are many different loss functions we might encounter (MSE, Cross-Entropy, KL-Divergence, huber, L1 (MAE), CTC, Triplet, ...) for different tasks.

The Cost Function (measure of discrepancy)

Recall from Foundations of Machine Learning:

- Mean Squared Error (MSE) loss for a single data point (here assumed to be a vector, but equally applicable to a scalar) is given by

  $\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sum_i (\hat{y}_i - y_i)^2 = (\hat{\boldsymbol{y}} - \boldsymbol{y})^\top (\hat{\boldsymbol{y}} - \boldsymbol{y})$
- We often multiply this by a constant factor of $\frac{1}{2}$ — can anyone guess/remember why?
- $\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ is the predominant choice for regression problems with linear activation in the last layer
- For a classification problem with Softmax or Sigmoidal (or really anything non-linear) activations, MSE can cause slow learning, especially if the predictions are very far off the targets
  - Gradients of ℓ_{MSE} are proportional to the difference in target and predicted multiplied by the gradient of the activation function[1]
  - The Cross-Entropy loss function is generally a better choice in this case

Binary Cross-Entropy

For the binary classification case:

$$\ell_{BCE}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

- The cross-entropy cost function is non-negative, $\ell_{BCE} > 0$
- $\ell_{BCE} \approx 0$ when the prediction and targets are equal (i.e. $y = 0$ and $\hat{y} = 0$ or when $y = 1$ and $\hat{y} = 1$)
- With Sigmoidal final layer, $\frac{\partial \ell_{BCE}}{\partial \boldsymbol{W}_i^{(2)}}$ is proportional to just the error in the output $(\hat{y} - y)$ and therefore, the larger the error, the faster the network will learn!
- Note that the BCE is the negative log likelihood of the Bernoulli Distribution

Binary Cross-Entropy — Intuition

- The cross-entropy can be thought of as a **measure of surprise**.
- Given some input x_i, we can think of $\hat{y}_i$ as the estimated probability that x_i belongs to class 1, and $1 - \hat{y}_i$ is the estimated probability that it belongs to class 0.
- Note the extreme case of infinite cross-entropy, if your model believes that a class has 0 probability of occurrence, and yet the class appears in the data, the 'surprise' of your model will be infinitely great.

[1] http://neuralnetworksanddeeplearning.com/chap3.html

Binary Cross-Entropy for multiple labels

In the case of multi-label classification with a network with multiple sigmoidal outputs you just sum the BCE over the outputs:

$$\ell_{BCE} = -\textstyle\sum_{k=1}^{K}[y_k \log(\hat{y}_k) + (1 - y_k)\log(1 - \hat{y}_k)]$$

where K is the number of classes of the classification problem, $\hat{y} \in \mathbb{R}^K$.

Numerical Stability: The Log-Sum-Exp trick

$$\ell_{BCE}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y)\log(1 - \hat{y})$$

- Consider what might happen early in training when the model might confidently predict a positive example as negative
  - $\hat{y} = \sigma(z) \approx 0 \implies z << 0$
  - if $\hat{y}$ is small enough, it will become 0 due to limited precision of floating-point representations
  - but then $\log(\hat{y}) = -\inf$, and everything will break!
- To tackle this problem implementations usually combine the sigmoid computation and BCE into a single loss function that you would apply to a network with linear outputs (e.g. `BCEWithLogitsLoss`).
- Internally, a trick called 'log-sum-exp' is used to *shift* the centre of an exponential sum so that only numerical underflow can potentially happen, rather than overflow[2].
  - Ultimately this means you'll always get a numerically reasonable result (and will avoid NaNs and Infs originating from this point).

Multiclass classification with Softmax Outputs

- Softmax can be thought of making the K outputs of the network mimic a probability distribution.
- The target label y could also be represented as a distribution with a single 1 and zeros everywhere else.
  - e.g. they are "one-hot encoded".
- In such a case, the obvious loss function is the *negative log likelihood* of the Categorical distribution (aka Multinoulli, Generalised Bernoulli, Multinomial with one sample)[3]: $\ell_{NLL} = -\sum_{k=1}^{K} y_k \log \hat{y}_k$
  - Note that in practice as y_k is zero for all but one class you don't actually do this summation, and if y is an integer class index you can write $\ell_{NLL} = -\log \hat{y}_y$.
- Analogously to what we saw for BCE, Log-Sum-Exp can be used for better numerical stability.
  - PyTorch combines LogSoftmax with NLL in one loss and calls this "Categorical Cross-Entropy" (so you would use this with a *linear output layer*)

Reminder: Gradient Descent

- Define total loss as $\mathcal{L} = \sum_{(\boldsymbol{x},y)\in \boldsymbol{D}} \ell(g(\boldsymbol{x}, \boldsymbol{\theta}), y)$ for some loss function ℓ, dataset $\boldsymbol{D}$ and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the **total loss** $\mathcal{L}$ by the learning rate η multiplied by the gradient: [1em] `for each Epoch:` $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$

[2] https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/

[3] Note: Keras calls this function 'Categorical Cross-Entropy'; you would need to have a Softmax output layer to use this

Reminder: Stochastic Gradient Descent

- Define loss function ℓ, dataset $\boldsymbol{D}$ and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Stochastic Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the loss of a **single item** ℓ by the learning rate η multiplied by the gradient: [1em] `for each Epoch:` `for each` $(\boldsymbol{x}, y) \in \boldsymbol{D}$`:` $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \ell$

A Quick Introduction to Tensors

Broadly speaking a tensor is defined as a linear mapping between sets of algebraic objects[4].

A tensor T can be thought of as a generalization of scalars, vectors and matrices to a single algebraic object.

We can just think of this as a multidimensional array[5].

- A $0D$ tensor is a scalar
- A $1D$ tensor is a vector
- A $2D$ tensor is a matrix
- A $3D$ tensor can be thought of as a vector of identically sized matrices
- A $4D$ tensor can be thought of as a matrix of identically sized matrices or a sequence of $3D$ tensors
- ...

Aside: Tensor Decompositions

- Just in the same way a matrix can be decomposed into a product of matrices (EVD, SVD, QR, LU, Cholesky, ...), there are tensor decompositions:
  - PARAFAC / Canonical polyadic / HO-SVD / Tucker
  - These have found their way into some deep learning models as a form of structural regularisation or weight reduction

Operations on Tensors in PyTorch

- PyTorch lets you do all the standard matrix operations on 2D tensors
  - including important things you might not yet have seen like the hadamard product of two $N \times M$ matrices: $\boldsymbol{A} \odot \boldsymbol{B}$)
- You can do element-wise add/divide/subtract/multiply to ND-tensors
  - and even apply scalar functions element-wise $(\log, \sin, \exp, ...)$
- you can slice, reshape, and *even index a single element* (**generally don't do that!**)
- PyTorch often lets you *broadcast* operations (just like in numpy)
  - if a PyTorch operation supports broadcast, then its Tensor arguments can be automatically expanded to be of equal sizes (without making copies of the data).[6]

[4] This statement is always entirely true

[5] This statement will upset mathematicians and physicists because its not always true for them (but it is for us!).

[6] Important - read and understand this after the lab next week: https://pytorch.org/docs/stable/notes/broadcasting.html

Tensors, batches and vectorisation

- The reality of training a model is that we neither use gradient descent or stochastic gradient descent; we do something in-between called mini-batch SGD.
- This works on batches of data (e.g. small subsets of the training set)
- These batches are assembled into a tensor
- Broadcasting is used to apply operations/functions to all the samples in the batch tensor *in parallel* to compute a loss vector
- the loss vector is summed/averaged using a *vectorised* method (e.g. `.sum()`)

Tensor implementation

It's important to understand something about how tensors are implemented in software and particularly how memory copies can be avoided...

We are siamese...

An important and clever trick:

Homework

PyTorch Tensor 101: `https://colab.research.google.com/gist/jonhare/d98813b2224dddbb234d2031510878e1/notebook.ipynb`