Train, Validate, Test

Recap of Basic Neural Networks

(and some Deep Network Fundamentals)

Jonathon Hare and Antonia Marcu

Vision, Learning and Control
University of Southampton

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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$.

Two Types of Supervised Learning

- Regression: The machine is asked predict k numerical values given some input. The machine is a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$.
- 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 $f : \mathbb{R}^n \rightarrow \{1, ..., k\}$.

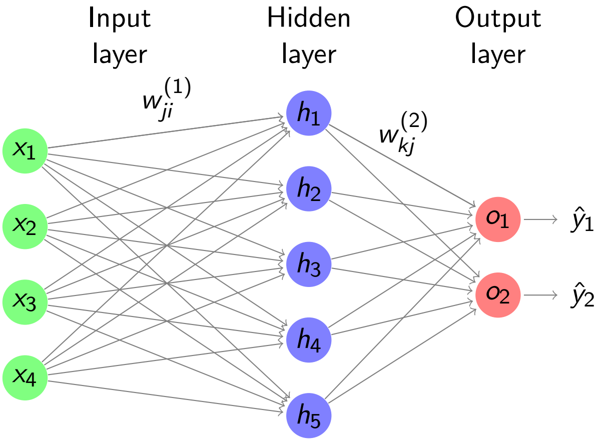
Two Types of Supervised Learning

- 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 $\hat{f} : \mathbb{R}^n \to \mathbb{R}^k$.

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

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

- What form should the final layer function g take?

Final layer activations

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

- What form should the final layer function g take?
- It depends on the task (and on the chosen loss function)...
  - regression $\rightarrow$ typically linear
  - binary classification $\rightarrow$ typically Sigmoid
  - multilabel classification $\rightarrow$ typically Sigmoid
  - multiclass classification $\rightarrow$ typically Softmax

Softmax

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

- Note that
  softmax makes reference to all the elements in the output.
- output: positive numbers that sum to 1.
- 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:
  1. the task (e.g. classification/regression/something else)
  2. the activation function of the last layer

Ok, so let's talk loss functions

- The choice of loss function depends on:
  1. the task (e.g. classification/regression/something else)
  2. the activation function of the last layer

  - For numerical reasons
    many times the activation is computed directly within the loss rather than being part of the model

Ok, so let's talk loss functions

- The choice of loss function depends on:
  1. the task (e.g. classification/regression/something else)
  2. the activation function of the last layer
     - For numerical reasons
       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
    for binary classification.

Ok, so let's talk loss functions

- The choice of loss function depends on:
  1. the task (e.g. classification/regression/something else)
  2. the activation function of the last layer
     - For numerical reasons
       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
    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 Loss Function (measure of discrepancy)

Recall from Foundations of Machine Learning:

- Mean Squared Error (MSE) loss for a single data point 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?

The Loss Function (measure of discrepancy)

Recall from Foundations of Machine Learning:

- Mean Squared Error (MSE) loss for a single data point 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

The Loss Function (measure of discrepancy)

Recall from Foundations of Machine Learning:

- Mean Squared Error (MSE) loss for a single data point 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 activations MSE can cause slow learning
  - Gradients of ℓ_{MSE} are proportional to the difference in target and predicted value, multiplied by the gradient of the activation function
  - **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 loss 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.

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:

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

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
  - $\hat{y} = \sigma(z) \approx 0 \implies z << 0$

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

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!

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`).

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

.

  - 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.

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".

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)
  - 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 $\ell_{NLL} = -\log \hat{y}_y$.

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)
  - 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 $\ell_{NLL} = -\log \hat{y}_y$.

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 λ

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 λ multiplied by the gradient:

for each Epoch:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{\theta}} \mathcal{L}$$

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 λ

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 λ multiplied by the gradient:

for each Epoch:

  for each $(\boldsymbol{x}, y) \in \boldsymbol{D}$:

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{\theta}} \ell$

A Quick Introduction to Tensors

Broadly speaking a tensor is defined as a linear mapping between sets of algebraic objects[1].

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[2].

- 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
- ...

---

[1] This statement is always entirely true

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

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 ($\log, \sin, \exp, \ldots$)
- 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).[3]

[3] Important - read and understand this after the lab: 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.

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)

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

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

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()`)

Homework

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

Watch and understand this:
https://southampton.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=c62809ad-af4d-4c7f-89e1-b26f00f85cd9