CS4619: Artificial Intelligence II

Training a Neural Network

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [4]:
%load_ext autoreload
%autoreload 2
%matplotlib inline

The autoreload extension is already loaded. To reload it, use:
%reload_ext autoreload

In [5]:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

In [6]:
from math import exp
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.datasets import mnist

In [7]:
# MNIST dataset

# Load MNIST into four Numpy arrays
(mnist_x_train, mnist_y_train), (mnist_x_test, mnist_y_test) = mnist.load_data()
mnist_x_train = mnist_x_train.reshape((60000, 28 * 28))
mnist_x_train = mnist_x_train.astype("float32") / 255

#Normalize
mnist_x_test = mnist_x_test.reshape((10000, 28 * 28))
mnist_x_test = mnist_x_test.astype("float32") / 255
Backpropagation

- We'll give an overview of Gradient Descent for neural networks
- Our goal is to get the 'flavour' of the algorithm — but you can ignore the details (non-examinable)
- Then, we will discuss the Vanishing Gradients Problem and its solutions (which is examinable)

Gradient Descent for Neural Networks

- As usual, we need the gradient of the loss function: the partial derivatives with respect to each of the weights $w$

  $$\frac{\delta J}{\delta w}$$

- Then we can update that weight in the usual way:

  $$w \leftarrow w - \alpha \frac{\delta J}{\delta w}$$

  where $\alpha$ is the learning rate
Notation

- The layers will be indexed from 0 to $L$.
  - So 0 is the input layer, 1 is the first hidden layer, … $L$ is the output layer.
- We will use superscripts in square brackets to refer to layers and subscripts to refer to neurons within layers.
  - E.g. $w_{[2,3]}^{[1]}$ is the weight from neuron 2 in layer 0 to neuron 3 in layer 1.
  - But we'll try to drop superscripts whenever they are obvious.
- We will use $g$ for an activation function and $g'$ for its derivative.
  - E.g. $z_3^{[1]}$ is the weighted sum for neuron 3 in layer 1.
  - As before $z$ is the input to a neuron (weighted sum).
  - E.g. $a_3^{[1]} = g(z_3^{[1]})$.
- We will use $\alpha$ for the output of a unit — the result of applying the activation function to the neuron's input.
  - E.g. $a_3^{[1]} = g(z_3^{[1]})$.
Partial derivatives

- We'll assume that the loss function is (half) of mean squared error and we'll consider just a single example, row vector \( x \)
- First consider a weight on a connection from a neuron in the last hidden layer to an output neuron, \( w_{i,j}^{[L]} \)
- We'll drop the \( L \) to clean things up, and anything that refers to \( j \) is also layer \( L \) and anything that refers to \( i \) is layer \( L - 1 \)
- The partial derivative of \( J \) with respect to \( w_{i,j} \)
  \[
  \frac{\delta J}{\delta w_{i,j}} = \frac{1}{2} (\hat{y}_j - y_j)^2 \\
  = \frac{1}{2} (a_j - y_j)^2 \\
  = a_i g'(z_j)(a_j - y_j)
  \]
- But let \( \Delta_j = g'(z_j)(a_j - y_j) \)
  - This quantity is called the error signal
  - It is \( \frac{\delta J}{\delta z_j} \), i.e. the change in loss caused by a change in \( z_j \) (the input to neuron \( j \))
- Then we can rewrite the derivative as just
  \[
  \frac{\delta J}{\delta w_{i,j}} = a_i \Delta_j
  \]
Partial derivatives
- Now consider a weight on a connection from a neuron in layer $L - 2$ to a neuron in the next hidden layer ($L - 1$), $w_{i,j}^{[L-1]}$.
- We'll drop the $L - 1$ to clean things up, and anything that refers to $j$ is also layer $L - 1$ and anything that refers to $i$ is layer $L - 2$.
- The partial derivative of $J$ with respect to $w_{i,j}$
  \[
  \frac{\delta J}{\delta w_{i,j}} = \frac{1}{2} \sum_{k \in L} (\hat{y}_k - y_k)^2 \\
  = \frac{1}{2} \sum_{k \in L} (a_k - y_k)^2 \\
  = a_i g'(z_j) \sum_{k \in L} g'(z_k)(a_k - y_k)w_{j,k} \\
  = a_i g'(z_j) \sum_{k \in L} \Delta_k w_{j,k}
  \]
- But now let $\Delta_j = g'(z_j) \sum_{k \in L} \Delta_k w_{j,k}$
  - Again, this is the error signal, but this time propagated back to layer $L - 1$.
  - It is $\frac{\delta J}{\delta z_j}$.
- Then we can rewrite the derivative as just
  \[
  \frac{\delta J}{\delta w_{i,j}} = a_i \Delta_j
  \]
- It is clear we can continue this for the weights between even earlier layers and clear too that there is a pattern here:
  - The derivative is always the error signal multiplied by the preceding activation value.
  - The error signals at one layer are based on the error signals of subsequent layers.
  - The error signals can be calculated by working backwards through the network.

The Backpropagation Algorithm
- **Random initialization**: initialize all weights randomly.
- **Forward propagation**: make predictions for all the training examples:
  - Layer by layer from from layer 1 to layer $L$:
    - Calculate the inputs to the units in that layer (weighted sums).
    - Calculate the outputs of the units in that layer (using activation function).
- **Backpropagation**:
  - Calculate the error signals at layer $L$.
  - Layer by layer in reverse from layer $L - 1$ to layer 1:
    - Calculate the error signals for the units in that layer.
- **Update all the weights**: $w_{i,j}^{[l]} \leftarrow w_{i,j}^{[l]} - \alpha \times a_i \times \Delta_{i,j}^{[l]}$

Demo
- We'll take a look at [http://experiments.mostafa.io/public/ffbpann/](http://experiments.mostafa.io/public/ffbpann/)
Class exercise

- Back prop starts by initializing weights randomly
  - E.g. it was common to use a normal distribution with mean of 0 and standard deviation of, e.g., 0.05
- A novice proposes instead to initialize them all to zero
- Why in general does this not make sense?

The Backpropagation Algorithm (Vectorized)

- **Random initialization**: initialize all weights randomly
- **Forward propagation**:
  - Calculate and store \( Z^{[1]} = XW^{[1]} \)
  - Layer by layer from layer 1 to layer \( L \):
    - Calculate and store \( A^{[l]} = g^{[l]}(Z^{[l]}) \)
    - Calculate \( Z^{[l+1]} = A^{[l]}W^{[l+1]} \)
    - Calculate and store \( G^{[l]} = g'(Z^{[l]})^T \)
- **Backpropagation**:
  - Calculate \( D^{[L]} = (A^{[L]} - Y)^T \)
  - Layer by layer in reverse from layer \( L = L - 1 \) to layer 1:
    - Calculate and store \( D^{[l]} = G^{[l]} * W^{[l]}D^{[l+1]} \)
- **Update all the weights**:
  - \( W^{[1]} = W^{[1]} - \alpha(D^{[1]}X)^T \)
  - \( W^{[l]} = W^{[l]} - \alpha(D^{[l]}A^{[l-1]})^T \) for all other values of \( l \)

Autodiff

- Whoa! We found the update rules for a particular network and a particular loss function
- We would get different update rules if we changed
  - the network, e.g. layers other than dense layers (batch normalization layers, convolutional layers, etc.)
  - the loss function
- Happily, we don't have manually to find the partial derivatives all over again
  - Neural networks consist of layers of operations, each with simple, known derivatives
  - The derivatives for the whole network can be obtained by repeated use of the **chain rule**:
    - To differentiate a function of a function, \( y = f(g(x)) \), let \( u = g(x) \) so that \( y = f(u) \), then
      \[
      \frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}
      \]
  - Hence, modern frameworks such as TensorFlow can compute gradients automatically in the backpropagation step
- This is known as **autodiff** (or, for the way it is used by backprop, **reverse mode autodiff**
The Vanishing Gradient Problem

- Each weight is updated by an amount proportional to the gradient of the loss function with respect to that weight
- But if the gradient is very small, the weight doesn't change much
  - This may prevent the network from converging to a good approximation of the target function
- We can now see that this is worse for deeper networks
  - The error signal becomes ever smaller as it is backpropagated
- We look at three solutions
  - Non-saturating activation functions
  - Better initialization
  - Batch normalization

Non-Saturating Activation Functions

- Certain activation functions, including the logistic (sigmoid) function, are one cause of the vanishing gradient problem

```python
In [8]:
def sigma(z):
    return 1 / (1 + exp(-z))

fig = plt.figure()
plt.title("Sigmoid function")
xvals = np.linspace(-10, 10, 21)
plt.plot(xvals, [sigma(xval) for xval in xvals], color = "red")
plt.xlabel("z")
plt.xlim(-10, 10)
plt.ylabel("$\sigma(z)$")
plt.ylim(0.0, 1.0)
plt.show()
```
When the input to this function becomes large (positive or negative), the function **saturates** (i.e. becomes very flat)

- When it saturates, its derivative is extremely close to 0 so there's not much gradient to propagate back to earlier layers (and what little gradient there is gets diluted as it propagates back)
- Even when the gradient is at its greatest (when input $z$ is 0 and $\sigma(z) = 0.5$), it is only 0.25
  - So in the back propagation, gradients always diminish by a quarter or more
- This is why we rarely use the logistic function as the activation function in the hidden layers of deep networks
- Lots of alternatives have been proposed, including the **rectified linear unit** activation function, ReLU

$$\text{ReLU}(z) = \max(0, z)$$

```python
In [9]: def relu(z):
    return max(z, 0)

fig = plt.figure()
plt.title("ReLU activation function")
xvals = np.linspace(-10, 10, 21)
plt.plot(xvals, [relu(xval) for xval in xvals], color = "red")
plt.xlabel("z")
plt.xlim(-10, 10)
plt.ylabel("$relu(z)$")
plt.ylim(0.0, 10.0)
plt.show()
```

- Neurons that use the ReLU activation function have obvious problems too:
  - If their input (weighted sum) is negative,
    - the output is zero
    - the gradient is zero
  - and if this is true for all examples in the training set then, in effect, the unit dies
  - The gradient changes abruptly at $z = 0$, which can make Gradient Descent bounce around
- Alternatives to ReLU such as Leaky ReLU, ELU (Exponential Linear Unit) and Scaled ELU have been proposed, having at least some non-zero gradient for negative inputs but they are slower to compute and they introduce further hyperparameters
- We'll stick with ReLUs in this module
Better Random Initialization

- It turns out that vanishing gradients are more likely for certain ways of initializing the weights.
- Perhaps the most typical method was to use a normal distribution with mean of 0 and standard deviation of, e.g., 0.05.
- Better methods have been proposed:
  - Glorot uniform initialization (also called Xavier uniform initialization).
  - Happily, this is the Keras default.

Batch Normalization

- We previously studied the usefulness of feature scaling when doing Gradient Descent for, e.g., linear regression.

... and we've been doing this to the features in our neural networks too.
- But, if this is a good idea for the inputs to the first hidden layer, why not use the same idea for the inputs to subsequent layers?
  - In other words, we normalize the activations (outputs) of layer \( l \) prior to them being used as inputs to layer \( l + 1 \).
  - It controls the distribution of the values throughout the training process.
- This, in essence, is the idea of batch normalization.

Some of the details

- (No need to learn these details — and there are some notation abuses anyway!)
- In summary, for a given layer, it standardizes the outputs of the neurons (subtract the mean, divide by the standard deviation), then it scales the result and adds an offset.
- It standardizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.

  - Let \( B \) be the batch of examples of size \( m_B \).
  - The mean of a batch of activation values \( \mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} a^{(i)} \).
  - Their standard deviation \( \sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (a^{(i)} - \mu_B)^2 \).
  - Standardize them: \( a_{\text{norm}}^{(i)} = \frac{a^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \), where \( \epsilon \) is a small value to avoid division-by-zero problems.

- But, while we want the different units to have comparable activation values (achieved by standardizing), we don’t necessarily want them to have mean 0 and standard deviation 1.
- So, we multiply by a scaling factor for the layer and add an offset for the layer.
  - Scale and add offset: \( \tilde{a}^{(i)} = \gamma a_{\text{norm}}^{(i)} + \beta \).
  - \( \gamma \) and \( \beta \) are also learned by the Gradient Descent.
Other benefits of Batch Normalization

- Batch Normalization reduces the vanishing gradients problem so much, we can even use saturating activation functions.
- Training becomes less sensitive to the method used for randomly initializing weights.
- Much larger learning rates work (faster convergence) with less risk of divergence.
- It acts like a regularizer.

Batch normalization in Keras

- Just add another layer!
- E.g.

```python
network.add(Dense, 12, activation="relu")
network.add(BatchNormalization())
```

This is how we will do batch normalization in CS4619.
- Ignore: in fact, there is some debate about whether we should batch normalize the activations of the layer (as above) or the weighted sum, before applying the activation function (as below):

```python
network.add(Dense, 12, activation="linear")
network.add(BatchNormalization())
network.add(Activation("relu"))
```

We'll stick with the former, which is more concise.

Exploring Vanishing Gradients with MNIST

```python
In [11]: def build_mnist_network(activation, initializer, use_batch_norm):
    network = Sequential()
    network.add(Dense(512, activation=activation, kernel_initializer=initializer, input_shape=(28 * 28,)))
    if use_batch_norm:
        network.add(BatchNormalization())
    network.add(Dense(10, activation="softmax", kernel_initializer=initializer))
    network.compile(optimizer=RMSprop(lr=0.003), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return network
```
In [12]:

```python
networks = [
    build_mnist_network("sigmoid", "random_normal", False),
    build_mnist_network("sigmoid", "random_normal", True),
    build_mnist_network("sigmoid", "glorot_uniform", False),
    build_mnist_network("sigmoid", "glorot_uniform", True),
    build_mnist_network("relu", "random_normal", False),
    build_mnist_network("relu", "random_normal", True),
    build_mnist_network("relu", "glorot_uniform", False),
    build_mnist_network("relu", "glorot_uniform", True)
]

for network in networks:
    network.fit(mnist_x_train, mnist_y_train, epochs=5, batch_size=128, verbose=0)
    test_loss, test_acc = network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
    print(test_acc)
```

0.9786 0.9913 0.982 0.9805 0.9783 0.9795 0.9747 0.9796

- We shouldn't read too much into the results above
- The ideas in this lecture apply to deep networks