Addition to Chapter 13

(Some portions of this document were AI generated and edited)

Useful information for training neural nets

We've focused on certain transfer (activation) functions, primarily the sigmoidal (logsig) and the ReLU. Another important function that acts on a layer of the neural net is the "softmax" function. This will be discussed below.

Additionally, probably the most popular optimizers for the neural net are "Adam" and "RMSProp", and we'll discuss those as well.

Finally, we'll summarize the forward and backward pass so that you can implement a simple feedforward net on your own.

The softmax function

When we're classifying functions, we typically use the columns of the identity matrix as our targets. However, the output of the neural net will not be 0's and 1's, but a continuum of real numbers. We could simply find which output cell is the maximum, and then convert the vector into all zeros except for that coordinate. However, we may want to analyze the output to try to determine what the neural net is responding to (or, what it is not responding to, if our error is not small enough).

Therefore, we want to convert a set of m real numbers into a set of m probabilities. If the output was all positive, then we could simply normalize the output. For example,

$$(3,1,5) \quad \Rightarrow \quad \left(\frac{3}{3+1+5}, \frac{1}{3+1+5}, \frac{5}{3+1+5}\right) = (1/3, 1/9, 5/9)$$

We could then set a value so that if the probabilities are all below that threshold, then the network is undecided, or we could make the maximum be 1, and the rest be zero- But we do now have the option of getting a better idea of what the neural net is doing.

What happens if all of the values in your network output are NOT all non-negative? This is where "softmax" really comes in (although softmax is normally employed even if the outputs are all positive).

Softmax will convert the numbers using the exponential function first (which maps them all to $(0, \infty)$), then normalizes the vector. Suppose our vector output is $(-1, 1, 3)$. Then softmax performs the following computations:

$$(-1,1,3) \quad \Rightarrow \quad \left(\frac{\mathrm{e}^{-1}}{\mathrm{e}^{-1}+\mathrm{e}^{1}+\mathrm{e}^{3}}, \frac{\mathrm{e}^{1}}{\mathrm{e}^{-1}+\mathrm{e}^{1}+\mathrm{e}^{3}}, \frac{\mathrm{e}^{1}}{\mathrm{e}^{-1}+\mathrm{e}^{1}+\mathrm{e}^{3}}\right)$$

In Python, we compute this as:

In Matlab, we would compute this as:

```
x=[-1 1 3];
out = exp(x)/sum(exp(x))
```

```
import numpy as np

x = np.array([-1, 1, 3])
out = np.exp(x) / np.sum(np.exp(x))
```

Which gives $(0.0158, 0.117, 0.867)$. In this case, we can say that the target is most likely $(0, 0, 1)$ (and we can put a probability on it).

We can think of the softmax function as an approximation to the argmax function (which returns a 1 where the vector has a max element, and zeros elsewhere).

Other optimization concerns

Adding Momentum to SGD

Plain SGD can be slow and unstable due to oscillations. Momentum is meant to help accelerate gradients in the right direction and dampen oscillations. Instead of updating parameters solely based on the current gradient, momentum accumulates a velocity vector that exponentially decays previously computed gradients:

$$v_t = \beta v_{t-1} + (1 - \beta)\nabla_\theta J(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha v_t$$

Where:

- θ_t is the parameter at iteration t (θ would typically be a weight or bias term of the neural net).
- α is the learning rate
- v_t is the velocity (momentum term). Rather than being the gradient vector itself, our velocity is a convex combination of the current gradient and the "velocity" used in the past.
- β is the momentum coefficient, typically around 0.9

To give you an idea of that convex combination, let's look at what happens for a one-dimensional function $y = f(x)$, starting at $x = a$, $\beta = 0.9$.

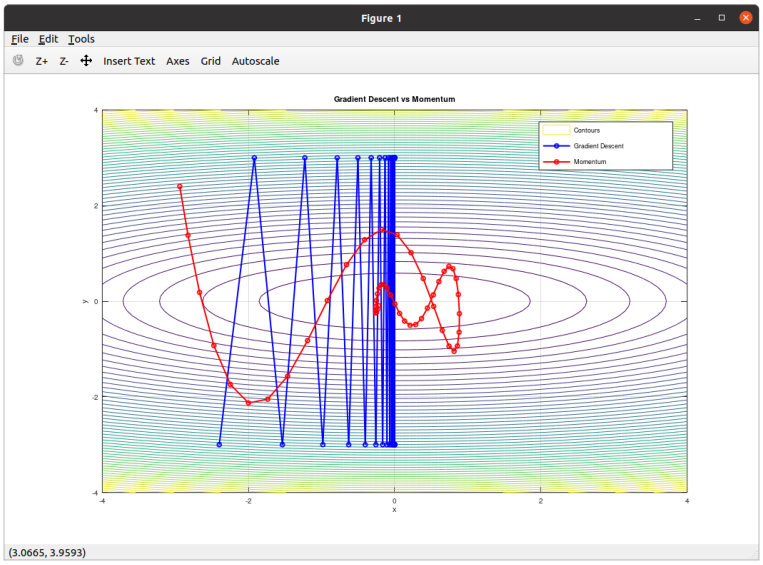
- Step 1: $v_1 = 0.9 \cdot 0 + 0.1 \cdot f'(a)$ and $a_1 = a - \alpha \cdot v_1$.
- Step 2: $v_2 = 0.9 \cdot v_1 + 0.1 \cdot f'(a_1)$ and $a_2 = a_1 - \alpha cdot v_2$.

And so on. We see that v_t is a quantity that "remembers" the past gradients.

As a numerical example, suppose we want to minimize $f(x, y) = x^2 + 10y^2$. Then with parameters:

- Learning rate: 0.1
- Starting point: $(-3, 3)$
- β for momentum: $\beta = 0.9$.
- For the momentum equation, set $v_0 = \vec{0}$.

In the plot below, we see the contour plots for f, and the trajectories. The blue trajectory which oscillates is regular gradient descent. The red trajectory which is smooth, is gradient descent with momentum. (The Matlab code for this example is attached at the end if you'd like the details).



RMSprop

RMSProp is short for "Root Mean Squared Propagation".

As we've seen, problems with gradient descent include:

- Oscillations in directions with steep gradients.
- Uniform learning rate for all parameters.

RMSprop addresses these issues by adapting the learning rate for each parameter based on the history of gradients. The way it does this is based on a **moving average of the squared gradients**. The notation you see below is common- The E stands for "expected value", which for us is the average. The g_t term is the gradient at step t.

RMSProp maintains an exponentially decaying average $E[g^2]_t$, updates the learning rate, and also scales the update. Also just to be clear, when you see a vector being squared, that means it is done element-wise.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t$$

where:

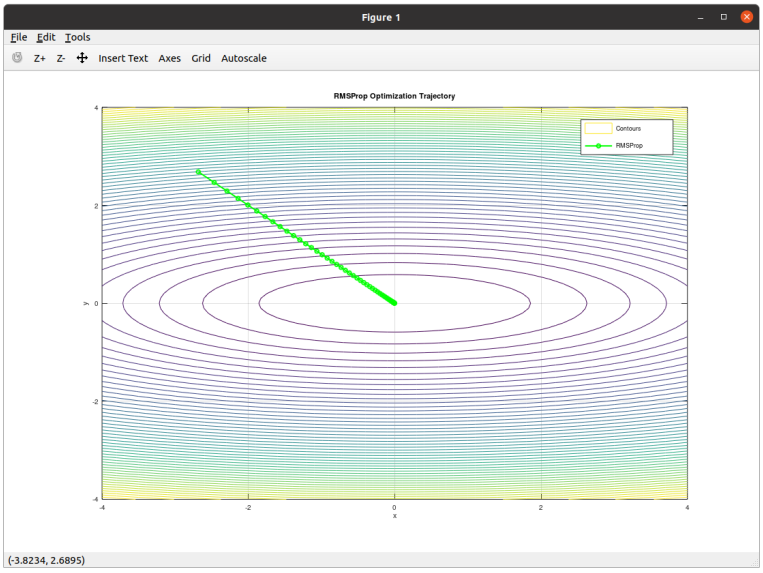
- $g_t = \nabla_\theta J(\theta_t)$ is the gradient at time t,
- γ is the decay rate (e.g., 0.9),
- ε is a small constant (e.g., 10^{-8}) to avoid division by zero.

This may look the same as gradient descent with momentum. Here's a comparison:

Feature	**Gradient Descent with Momentum**	**RMSProp**
Idea	Uses a moving average of past gradients to accelerate convergence	Adjusts the learning rate adaptively based on a moving average of squared gradients
Memory Term	Velocity: $v_t = \beta v_{t-1} + (1-\beta)\nabla J$	Accumulator: $s_t = \beta s_{t-1} + (1-\beta)(\nabla J)^2$
Update Equation	$\theta = \theta - \alpha v_t$	$\theta = \theta - \alpha \frac{\nabla J}{\sqrt{s_t + \epsilon}}$
Effect on Learning Rate	Uses a fixed global learning rate α	Uses an adaptive per-parameter learning rate
Behavior	Helps reduce oscillations and accelerates convergence in consistent directions	Normalizes updates to prevent too large steps in steep directions
Best For	Problems with ravines or consistent gradient directions	Problems with varying gradient magnitudes or noisy gradients

Table 1: Comparison of Gradient Descent with Momentum vs RMSProp

Here we have the same function as before, $z = f(x, y) = x^2 + 10y^2$. The full Matlab code is attached at the end, but here is the resulting trajectory. There is a stark difference between this trajectory and the previous ones.



Adam

Adam (short for Adaptive Moment Estimation) combines Momentum and RMSprop. It computes two moving averages:

- The first moment (mean): m_t
- The second moment (uncentered variance): v_t

The update rules are:

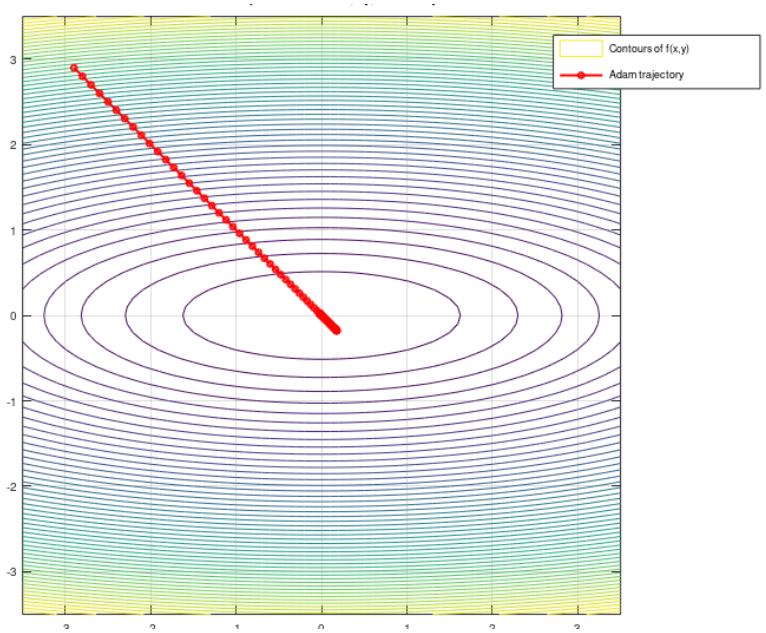
$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t
\end{aligned}
$$

To give a sense for how this works, here are some values of the training parameters that tend to work as default values:

- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\varepsilon = 10^{-8}$

- $\eta = 0.001$

And to compare the trajectory to our previous examples, there is Matlab code attached that used Adam as the optimizer for $f(x, y) = x^2 + 10y^2$, and here is the resulting trajectory.



In conclusion, Adam is a powerful optimizer that adapts both the step size and the direction of updates based on gradient history. It is particularly effective for complex and noisy training problems, making it a go-to choice for optimizing neural nets.

- Combines benefits of Momentum and RMSprop.
- Requires little memory.
- Automatically adjusts learning rates for each parameter.
- Works well in practice for a wide range of models.

Attachment 1: Matlab code for Momentum Example

```
% Script:  Gradient Descent with Momentum
% Function definition
f = @(x, y) x.^2 + 10*y.^2;
grad = @(x, y) [2*x; 20*y];

% Parameters
alpha = 0.1;
num_iters = 50;


% ----- Gradient Descent with Momentum -----
x_m = -3; y_m = 3;
v = [0; 0];
beta = 0.9;
traj_mom = zeros(2, num_iters);

for i = 1:num_iters
    g = grad(x_m, y_m);
    v = beta * v + (1 - beta) * g;
    x_m = x_m - alpha * v(1);
    y_m = y_m - alpha * v(2);
    traj_mom(:, i) = [x_m; y_m];
end

% ----- Regular Gradient Descent -----
x_g = -3; y_g = 3;
traj_gd = zeros(2, num_iters);

for i = 1:num_iters
    g = grad(x_g, y_g);
    x_g = x_g - alpha * g(1);
    y_g = y_g - alpha * g(2);
    traj_gd(:, i) = [x_g; y_g];
end

% ----- Plotting -----
[xs, ys] = meshgrid(-4:0.1:4, -4:0.1:4);
zs = f(xs, ys);

figure;
contour(xs, ys, zs, 50); hold on;
```

```
plot(traj_gd(1, :), traj_gd(2, :), 'b-o', 'LineWidth', 2);
plot(traj_mom(1, :), traj_mom(2, :), 'r-o', 'LineWidth', 2);
title('Gradient Descent vs Momentum');
xlabel('x'); ylabel('y');
legend('Contours', 'Gradient Descent', 'Momentum');
grid on;
```

Attachment 2: RMSProp Example in Matlab

```
% Function and gradient
f = @(x, y) x.^2 + 10*y.^2;
grad = @(x, y) [2*x; 20*y];

% Parameters
alpha = 0.1;          % Base learning rate
beta = 0.9;           % Decay rate for moving average
epsilon = 1e-8;
num_iters = 50;
RMSProp is short for ‘‘Root Mean Squared Propagation’’.
% Initialization
x_r = -3; y_r = 3;
s = [0; 0];           % Running average of squared gradients
traj_rms = zeros(2, num_iters);

% RMSProp loop
for i = 1:num_iters
    g = grad(x_r, y_r);
    s = beta * s + (1 - beta) * (g.^2);  % element-wise square
    x_r = x_r - alpha * g(1) / (sqrt(s(1)) + epsilon);
    y_r = y_r - alpha * g(2) / (sqrt(s(2)) + epsilon);
    traj_rms(:, i) = [x_r; y_r];
end

% ---- Plot with contours and comparison ----
[xs, ys] = meshgrid(-4:0.1:4, -4:0.1:4);
zs = f(xs, ys);
RMSProp is short for ‘‘Root Mean Squared Propagation’’.
figure;
contour(xs, ys, zs, 50); hold on;
plot(traj_rms(1, :), traj_rms(2, :), 'g-o', 'LineWidth', 2);
title('RMSProp Optimization Trajectory');
xlabel('x'); ylabel('y');
legend('Contours', 'RMSProp');
grid on;
```

Attachment 3: Sample Code for Adam

```
% Adam Optimization for f(x, y) = x^2 + 10y^2 with 2D contour plot

% Objective function and its gradient
f = @(x, y) x.^2 + 10*y.^2;
grad_f = @(x, y) [2*x; 20*y];

% Initialize variables
x = -3;
y = 3;
theta = [x; y];

% Adam hyperparameters
alpha = 0.1;      % Learning rate
beta1 = 0.9;
beta2 = 0.999;
epsilon = 1e-8;

% Initialize moments
m = [0; 0];
v = [0; 0];

% Number of iterations
max_iters = 100;
history = zeros(max_iters, 3);  % Store x, y, and f(x,y)

for t = 1:max_iters
    % Compute gradient
    g = grad_f(theta(1), theta(2));

    % Update biased first moment estimate
    m = beta1 * m + (1 - beta1) * g;

    % Update biased second raw moment estimate
    v = beta2 * v + (1 - beta2) * (g.^2);

    % Compute bias-corrected moments
    m_hat = m / (1 - beta1^t);
    v_hat = v / (1 - beta2^t);

    % Update parameters
    theta = theta - alpha * m_hat ./ (sqrt(v_hat) + epsilon);
```

```
    % Store for plotting
    history(t, :) = [theta(1), theta(2), f(theta(1), theta(2))];
end

% Display final result
fprintf('Minimized at x = %.4f, y = %.4f, f(x,y) = %.4f\n', ...
        theta(1), theta(2), f(theta(1), theta(2)));

% Generate grid for contour plot
[x_grid, y_grid] = meshgrid(-3.5:0.1:3.5, -3.5:0.1:3.5);
z_grid = f(x_grid, y_grid);

% Plot the contour and optimization path
figure;
contour(x_grid, y_grid, z_grid, 50); hold on;
plot(history(:,1), history(:,2), 'ro-', 'LineWidth', 2, 'MarkerSize', 5);
xlabel('x'); ylabel('y');
title('Adam Optimization on f(x,y) = x^2 + 10y^2');
grid on; axis equal;
legend('Contours of f(x,y)', 'Adam trajectory');
```