Autograd for Automatic Differentiation and for Auto-Construction of Computational Graphs

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Sunday 28th January, 2024 09:48

Preamble

To TOC

Consider a machine-learning-based system that figures out its critical parameters by minimizing a cost function derived from the prediction errors it makes on the training data. In such a system, the learnable parameters define a hyperplane and the cost function sits as a surface over the hyperplane. The learning in such a system consists of identifying the point in the hyperplane where the cost function takes its smallest value. [Note that the height of the cost function over any point in the parameter hyperplane is a measure of the prediction error that the system would make when the parameters are set to that point.]

A classic algorithm for finding the point in the parameter hyperplane where the cost function is globally minimum is the Gradient Descent (GD) algorithm. GD is iterative: You start out by randomly choosing a point in the parameter hyperplane, compute the partial derivatives of the cost-function surface at that point, and then use those partial derivatives to take a step in the direction of the goal point (that is, the point under the global minimum). You then iterate with the new point.

Although GD is numerically robust, it is not possible to use it directly on account of its propensity to get stuck in local minima and its slow convergence as it approaches the solution point.

Preamble (contd.)

A practical alternative to GD is **Stochastic Gradient Descent (SGD)** that is less likely to get stuck in a local minimum.

SGD is based on a critical intuition acquired from the formulation of GD solutions: that the cost function surface used for GD is a function of both the learnable parameters and the training data being used for making the predictions.

Since GD uses all the training data all at once at each iteration of the algorithm, its cost function surface remains fixed during the iterations.

On the other hand, SGD does its updating of the learnable parameters by using only a small batch of randomly drawn training samples at each iteration. So even if the current solution point in the parameter hyperplane is at a local minimum in the cost-function surface corresponding to the current batch, **it would be highly unlikely that the same point would be a local minimum for the cost-function surfaces corresponding to the future randomly drawn batches of samples**. What that implies is that the solution path can keep on growing with SGD as more and more training data is ingested.

Preamble (contd.)

With that intro to SGD, we'll be ready to talk about the PyTorch's all-important Autograd module for implementing SGD. **You could say that Autograd is at the heart of the PyTorch platform.**

Autograd's main job is to estimate the partial derivatives of each learnable parameter in a layer with respect to the output of the layer **during the forward propagation of the training data through the network**. These partial derivatives are stored in what's known as a **dynamic** computational graph (CG).

Subsequently, when the loss calculated at the output of a neural network is backpropagated, the computational graph facilitates correct chaining of the partial derivatives for updating the value of each learnable parameter.

When I say that Autograd constructs a *dynamic* computational graph, what that means is that a new graph is constructed **for each new batch of training data** coursing its way through the network. An alternative to a dynamic computational graph is a *static* computational graph that is used in the TensorFlow platform for deep learning.

Preamble (contd.)

To make it easier for a student new to DL to understand the ideas mentioned on the last couple of slides, I have created a Python module called `ComputationalGraphPrimer` (CGP) that's available at

`https://pypi.org/project/ComputationalGraphPrimer/`

The `Examples` directory of CGP contains the following two scripts that are based on handcrafted neural networks:

```
one_neuron_classifier.py
multi_neuron_classifier.py
```

The functions invoked by these scripts illustrate the following: **(1)** How one can compute the partial derivatives of the loss with respect to the learnable parameters during the forward propagation of a batch of input data; **(2)** How those partial derivatives can subsequently be used for updating the learnable parameters during backpropagation of loss; and **(3)** The batch based smoothing as required by SGD of the partial derivatives and the predictions errors before the parameters are updated. The partial derivatives of the output of a neural layer with respect to each element of the input to the layer are based on the analytical formulas shown in the last section of this lecture.

Preamble (contd.)

The Examples directory of CGP also contains the following Torch.nn based script

```
verify_with_torchnn.py
```

as a check on the performance of the handcrafted networks used in the previously mentioned two scripts. You'll get superior results (in terms of how fast the loss decreases with training iterations) with the Torch.nn based networks on account of the step-size optimizer used. That then serves as the basis for the homework associated with this lecture — to incorporate a rudimentary optimizer in the step-size calculation code for the handcrafted examples.

CGP also illustrates dataflow in a general DAG (Directed Acyclic Graph) of nodes, both the forward propagation of the input data, the computation of the partial derivatives during the forward propagation of data, and the backpropagation of loss. The goal of this illustration is only to introduce a student to the notion of dataflows and computations in a DAG. You will see references to DAG in the official PyTorch documentation on the Autograd module. CGP's calculations of the partial derivatives in a DAG are based on the finite difference method.

Preamble (contd.)

Also included in CGP is a demonstration of how to extend the Autograd class for more specialized processing of the data that may require remembering certain designated attributes of the data as it is flowing in the forward direction and using the remembered values to make changes to the loss flowing backwards during backpropagation.

Outline

1. Revisiting Gradient Descent 9
2. Problems with a Vanilla Application of GD 19
3. Stochastic Gradient Descent 21
4. The Computational Graph Primer (CGP) 32
5. CGP Demo 1: `graph_based_dataflow.py` 39
6. CGP Demo 2: `one_neuron_classifier.py` 52
7. CGP Demo 3: `multi_neuron_classifier.py` 63
8. CGP Demo 4: `verify_with_torchnn.py` 83
9. Autograd 90
10. Extending Autograd 94
11. Step Size Optimization for SGD 106

Outline

1. Revisiting Gradient Descent 9
2. Problems with a Vanilla Application of GD 19
3. Stochastic Gradient Descent 21
4. The Computational Graph Primer (CGP) 32
5. CGP Demo 1: `graph_based_dataflow.py` 39
6. CGP Demo 2: `one_neuron_classifier.py` 52
7. CGP Demo 3: `multi_neuron_classifier.py` 63
8. CGP Demo 4: `verify_with_torchnn.py` 83
9. Autograd 90
10. Extending Autograd 94
11. Step Size Optimization for SGD 106

Loss Surfaces and the Parameter Hyperplane

A good place to start this lecture is by reviewing what you have already learned from a previous lecture by Professor Bouman.

- Let's say our training data consist of the pairs $(\mathbf{x}_i, y_i)$ where each vector $\mathbf{x}_i, i = 1, \ldots, m$, is a multidimensional measurement of the observed data for the i^{th} training sample and y_i the corresponding ground-truth classification label. Given, say, 10 classes, the value of each y_i will be an integer from the set $\{0, \ldots, 9\}$.
- Let the predicted label for training sample $\mathbf{x}_i$ be the integer y_i'.
- We can measure the loss for each sample by, say, the mean square error:

$$L_i \quad = \quad (\mathbf{y}_i - \mathbf{y'}_i)^2 \tag{1}$$

[This would actually be the **DUMBEST WAY** to measure loss in any classification network. It should not make any difference in loss if a training sample whose true class is 1 is misclassified as 2 or as 9 — unless for some reason you deliberately want to penalize differently the different types of misclassifications. The 'standard' way to measure loss in classification networks is the Cross-Entropy Loss that I'll be presenting in detail in an upcoming lecture. Nonetheless, the formula shown above makes for a more compact explanation of the main points I need to convey here.]

Loss Surfaces and the Parameter Hyperplane (contd.)

- We can now measure the total loss for all *m* training samples by

$$L \;=\; \sum_{1}^{m} L_i \;=\; \sum_{1}^{m} (\mathbf{y}_i - \mathbf{y'}_i)^2 \tag{2}$$

- If the vector **p** represents all of the learnable parameters of the system, each prediction $\mathbf{y'}_i$ will be a function of **p** and also of the input data sample $\mathbf{x}_i$. So we can say:

$$\mathbf{y'}_i \;=\; \mathbf{f}_i(\mathbf{x}_i, \mathbf{p}) \tag{3}$$

- We can now write for the overall loss:

$$L \;=\; \sum_{1}^{m} L_i \;=\; \sum_{1}^{m} \Big(\mathbf{y}_i - \mathbf{f}_i(\mathbf{x}_i, \mathbf{p})\Big)^2 \tag{4}$$

Loss Surfaces and the Parameter Hyperplane (contd.)

- Since the overall loss is the sum of the squares of the error in each training sample separately, we are allowed to represent the summation over all the training samples by

$$L \quad = \quad ||\mathbf{Y} - \mathbf{F}(\mathbf{X}, \mathbf{p})||^2 \tag{5}$$

  where you can think of **Y** as us lining up all the ground-truth integer labels for all the *m* training samples into one large vector.

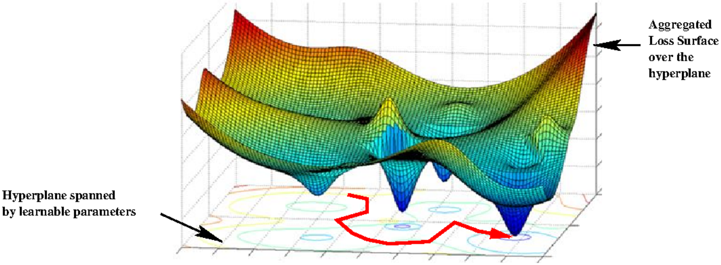
- By the same token, **F** also represents us lining up all the predictions for the individual samples into a large vector whose size is same as that of **Y**.

NOTE: It is important that you convince yourself that Eq. (4) can be expressed in the form shown in Eq. (5). Construct for yourself a small example involving two sets of vectors $\{\vec{s}_1, \vec{s}_2, \vec{s}_3\}$ and $\{\vec{t}_1, \vec{t}_2, \vec{t}_3\}$ with each $\vec{t}_i$ being an approximation to $\vec{s}_i$. A straightforward formula for the overall approximation error would be $\sum_{i=0}^{3} ||\vec{s}_i - \vec{t}_i||^2$. Now let's say $\vec{s}_i = \begin{pmatrix} s_{i,1} \\ s_{i,2} \end{pmatrix}$ and $\vec{t}_i = \begin{pmatrix} t_{i,1} \\ t_{i,2} \end{pmatrix}$. The overall error is now given by $(s_{1,1} - t_{1,1})^2 + (s_{1,2} - t_{1,2})^2 + (s_{2,1} - t_{2,1})^2 + (s_{2,2} - t_{2,2})^2 + (s_{3,1} - t_{3,1})^2 + (s_{3,2} - t_{3,2})^2$. Let's now define two vectors $S = (s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}, s_{3,1}, s_{3,2})^{tr}$ and $T = (t_{1,1}, t_{1,2}, t_{2,1}, t_{2,2}, t_{3,1}, t_{3,2})^{tr}$ where the superscript *tr* stands for 'transpose' since we want column vectors. The overall error may now be expressed simply as $||S - T||^2$.

Loss Surfaces and the Parameter Hyperplane (contd.)

- The quantity **X** in the formula shown above now represents all the training data.
- The only unknowns in the expression for overall loss are the learnable parameters in the vector p. L is obviously a surface of some sort over the hyperplane spanned by the elements of p.
- This surface is best visualized as shown on the next slide.
- Our job is to find that point in the parameter hyperplane where loss surface has the least value.

Visualizing the Loss Surface



The 3D plot shown in the figure is from the following source: `https://www.fromthegenesis.com/gradient-descent-part1/`

Gradient Descent for Finding the Best Point in the Parameter Hyperplane

- The thing to bear in mind about the height of the surface at each point in the parameter space is that L is scalar, no different from any other scalar like, say, the temperature of the air at each point in a classroom.

- To understand the dependence of a scalar on all its defining parameters, you examine the gradient of the scalar, which is a vector that consists of the partial derivatives of the scalar with respect to each of the parameters. We denote the gradient of L at each point in the hyperplane by $\frac{\delta L}{\delta \mathbf{p}}$.

- **The gradient of a scalar is a vector that, in the parameter hyperplane, always points in the direction in which the scalar is increasing at the fastest rate.** Our goal in gradient descent (GD) is to head in the opposite direction since we want to reach the minimum.

Using GD Means Having to Calculate the Jacobian

- Based on the presentation made so far, you can imagine what if we are currently at point $\mathbf{p}_k$ in the parameter space, our next point in our journey to the global minimum would be at

$$\mathbf{p}_{k+1} \quad = \quad \mathbf{p}_k \quad - \quad \alpha \cdot \frac{\delta L}{\delta \mathbf{p}} \tag{6}$$

  where α is a small number that is called the **learning rate**.

- So using GD boils down to estimating gradient vector $\frac{\delta L}{\delta \mathbf{p}}$. If you take a derivative of L with respect to $\mathbf{p}$, one arrives at the following formula for the gradient:

$$\frac{\delta L}{\delta \mathbf{p}} \quad = \quad 2J_\epsilon^T(\mathbf{p})\epsilon(\mathbf{p}) \tag{7}$$

  where $J_\epsilon(\mathbf{p})$ is the Jacobian and where ϵ is the prediction error given by $\epsilon = \mathbf{Y} - \mathbf{F}(\mathbf{X}, \mathbf{p})$.

Using GD Means Having to Calculate the Jacobian (contd.)

- Since the vector $\mathbf{Y}$ in $\epsilon = \mathbf{Y} - \mathbf{F}(\mathbf{X}, \mathbf{p})$ consists of the ground-truth class labels, $\mathbf{Y}$ is a constant from the standpoint of differentiation. Therefore, we can write $J_\epsilon(\mathbf{p}) = -J_\mathbf{F}(\mathbf{p})$. Substituting this in our update formula, we get

$$\mathbf{p}_{k+1} \quad = \quad \mathbf{p}_k \quad + \quad 2 \cdot \alpha \cdot J_\mathbf{F}(\mathbf{p}_k) \cdot \epsilon_k \tag{8}$$

- The Jacobian $J_\mathbf{f}(\mathbf{p})$ is given by

$$J_\mathbf{F}(\mathbf{p}) \quad = \quad \begin{bmatrix} \frac{\delta f_1}{\delta p_1} & \dots & \frac{\delta f_1}{\delta p_n} \\ \vdots & \vdots & \vdots \\ \frac{\delta f_m}{\delta p_1} & \dots & \frac{\delta f_m}{\delta p_n} \end{bmatrix} \tag{9}$$

[**Good to commit to memory:** Each row of the Jacobian is a partial derivative of each prediction with respect to all n learnable parameters. For simplicity in notation, we are pretending that the predicted class label for each training image is a scalar number. You could assume that we represent the class labels by distinct integers.]

Using GD Means Having to Calculate the Jacobian (contd.)

- One of my reasons for explicitly defining the Jacobian in these slides is to help you understand the PyTorch's main documentation page on Autograd at:

  `https://pytorch.org/docs/stable/autograd.html`

- Now you know the concept of the Jacobian, that it is a matrix, and that the number of columns in this matrix is equal to the number of learnable parameters, and that the number of rows is equal to the number of training samples you want to process at a time.

- As you will soon see, the phrase "*the number of training samples you want to process at a time*" is critical as it leads to the notion of a "batch" in how the training is carried out in deep networks.

Outline

Problems with a Vanilla Application of GD

As Professor Bouman has already pointed out, there are several problems with a straightforward application of GD:

- It can be very slow to converge — especially as the solution point gets closer and closer to the optimum (because the gradients will approach zero).

- Its propensity to get stuck in a local minimum.

- When the loss surface is a long narrow valley in the vicinity of the final solution, the solution path will oscillate around the bottom "spine" of the valley as it slowly approaches the optimum.

[**NOTE:** In order to fix the slow convergence problem, the more traditional practical applications of GD involve combining GD with the estimation of Gauss-Newton (GN) jumps at each iteration. A GN jump can potentially get to the destination in one fell swoop, but it is numerically unstable. For that reason, a GN increment to the solution path is accepted only after it is evaluated for its quality. A commonly used combination GD and GN is known as the Levenberg-Marquardt (LM) algorithm. This is one of the most famous algorithms that is used in computer vision for solving all sorts of cost minimization problems. A numerically fast version of this algorithm is known as Bundle Adjustment. LM involves inverting the matrix product $J_{\mathbf{f}}^T J_{\mathbf{f}}$ which makes it infeasible for DL where you may have millions of learnable parameters.]

Outline

Stochastic Gradient Descent (SGD)

- In DL networks, the learnable parameters are estimated with a powerful variant of GD that is known as the Stochastic Gradient Descent. With SGD, the solution path is less likely to get stuck in a local minimum (which is a perennial problem with GD).

- The main idea in SGD is that you only process a small number of training samples at a time — the number may be as few as one, but you are likely to get better results if the number is greater than one.

- A re-examination of the previous slides would show that everything we have done so far holds regardless of the value of m, the number of training samples.

- **So let's say you have a large training dataset, is there anything to be gained by only using a small number of training samples at a time?**

SGD (contd.)

- The question is what happens to our visualization of the solution path if only use small values like 4 or 5 for *m* at each iteration of training? Recall that *m* is the number of training samples in Eqs. (2) through (5).

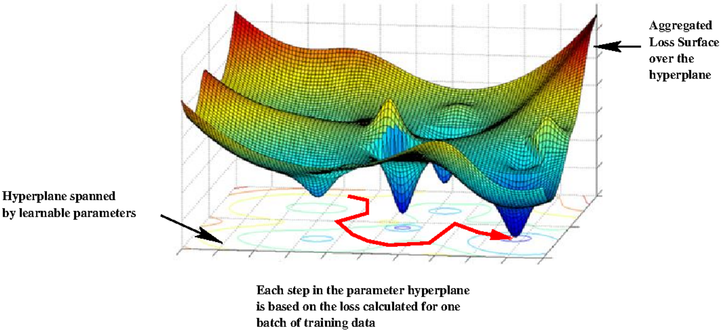
- **A most important point here is that the cost-function surface over the parameter hyperplane depends on the training data itself. When you ingest all of the data all at once, as is the case with GD, the surface remains fixed as you search for the point in the hyperplane where the cost has the globally minimum value.**

- However, if you process the training data only *m* samples at a time, you'll be changing the shape of the surface. **As a result, it is highly unlikely that a local valley in the cost-function surface for the previously processed *m* samples would remain at exactly the same location in the parameter hyperplane for all future groupings of *m* samples.**

SGD (contd.)

- Another way of arguing the point made on the previous slide would be that the parameter hyperplane would generally of very high dimensionality (think millions) and any given small set of training samples will be sensitive to just a small subset of all the parameters.
- For example, in a classification network, the different parameters will correspond to the different features whose presence determines the final classification decision on the input image. Therefore, the path increments $\delta\mathbf{p}$ in the parameter hyperplane with different small sets of training samples are likely to point in different directions.
- We can therefore expect there to be a bit of “chaos” associated with the path extensions if we only consider a small number of training samples at a time. *As it turns out, there is experimental evidence that this chaos helps the solution path to jump out of what would otherwise be local minima in the cost function surface if we were to process all of the data all at once.* See the visualization on the next slide.

Visualizing the Loss Surface (again)



In SGD, each path increment in the parameter hyperplane is now determined by just the batch-size number of training samples. But note that the loss surface will now be different for each batch — since the surface depends on the training data.

The 3D plot from: `https://www.fromthegenesis.com/gradient-descent-part1/`

Computation of the Derivatives Required by the Jacobian

- We will revisit Eqs. (8) and (9) shown earlier:

$$\mathbf{p}_{k+1} \quad = \quad \mathbf{p}_k + 2 \cdot \alpha \cdot J_{\mathbf{F}}(\mathbf{p}_k) \cdot \epsilon_k \tag{10}$$

  where the Jacobian $J_{\mathbf{f}}(\mathbf{p})$ is given by

$$J_{\mathbf{F}}(\mathbf{p}) \quad = \quad \nabla_{\mathbf{p}} \mathbf{F} \quad = \quad \begin{bmatrix} \frac{\delta f_1}{\delta p_1} & \ldots & \frac{\delta f_1}{\delta p_n} \\ \vdots & \vdots & \vdots \\ \frac{\delta f_m}{\delta p_1} & \ldots & \frac{\delta f_m}{\delta p_n} \end{bmatrix} \tag{11}$$

  where $\nabla_{\mathbf{p}}\mathbf{F}$ stands for the partial derivatives of the elements of the prediction vector **F** with respect to the individual elements in the parameter vector **p** — which is what the Jacobian is.

- Obviously, the path increment given by Eq. (10) can only be calculated after you have seen the prediction error ϵ_k for the k^{th} step. However, Eq. (10) allows for the following style of computation: **the partial derivatives $\nabla_{\mathbf{p}}\mathbf{F}$ can be calculated independently even before we have gotten hold the prediction error ϵ_k. This observation will prove foundational to how learning takes place in a neural network.**

Computation of the Derivatives Required by the Jacobian (contd.)

- To see what's made possible by the observation at the bottom of the previous slide, consider first the simple case of a single-layer neural network whose input/output relationship is described by

$$\mathbf{y} \quad = \quad F(\mathbf{x}, \mathbf{p}) \quad = \quad g(\mathbf{W} \cdot \mathbf{x}) \tag{12}$$

  where **W** is the link matrix between the input and the hidden layer, $\mathbf{W} \cdot \mathbf{x}$ the pre-activation output at the hidden layer, **and $g()$ the activation function**. [The dot operator in $\mathbf{W} \cdot \mathbf{x}$ is meant to emphasize the role of **W** as an operator.]

- In Eq. (12), we can refer to $\mathbf{W} \cdot \mathbf{x}$ as the pre-activation output from the neural layer and $g(.)$ as the post-activation output. We can now write

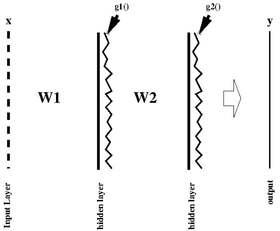
$$\nabla_{\mathbf{p}} F \quad = \quad \partial_g \left(\nabla_{\mathbf{W}} (\mathbf{W} \cdot \mathbf{x}) \right) \tag{13}$$

  where what you see on the right-hand-side follows from the chain rule. That is, we express the partial of the prediction vector $F()$ with respect to the parameters in **p** as a chain in which the first partial is for the output of $g()$ with respect to the pre-activation output, followed by the Jacobian of the latter with respect to the parameter vector **p**.

Computation of the Derivatives Required by the Jacobian (contd.)

- Let's now bring in one more hidden layer into this network. Let $\mathbf{W}_1$ and $\mathbf{W}_2$ be the link matrices between the input and the first hidden layer and between the first hidden layer and the next hidden layer. If we represent the output of the first hidden layer by $\mathbf{h}$, we can now write

$$\mathbf{y} \quad = \quad F(\mathbf{x}, \mathbf{p}) \quad = \quad g_2(\mathbf{W}_2\mathbf{h}) \quad = \quad g_2(\mathbf{W}_2 \cdot g_1(\mathbf{W}_1 \cdot \mathbf{x})) \tag{14}$$



Computation of the Derivatives Required by the Jacobian (contd.)

- Taking the partials of both sides for the case of two hidden layers shown on the previous slide, we get

$$\nabla_{\mathbf{p}} F \quad = \quad \partial_{g_2}(\nabla_{\mathbf{W}_2}(\mathbf{W}_2 \cdot (\partial_{g_1}(\nabla_{\mathbf{W}_1}(\mathbf{W}_1 \cdot \mathbf{x}))))) \tag{15}$$

- Let's now extend the observation made at the bottom of Slide 26 to the chain of partials shown above: **As the training data flows in the forward direction from layer to layer, we can apply that observation to each layer independently, thinking of the F vector in that equation at the output of each layer as the predictions to be expected from that layer.** That would allow us to calculate the partial derivatives during the forward propagation of the data — despite the fact that we have no idea regarding the prediction error at the output of the final layer.

- Subsequently, after we have observed the real prediction error at the output of the overall network, we can backpropagate it through the network, layer to layer, and update the parameters in accordance with Eq. (10).

Computation of the Derivatives Required by the Jacobian (contd.)

- The derivation shown on last few slides was for the very simple case involving what's referred to a "single-channel" input — meaning that we could think of the input $\mathbf{x}$ as a vector — and no biases, let alone the absence of any convolutional processing of the data.

- The more general case would involve multi-channel inputs and multi-channel outputs for each layer. This requires generalizing our derivatives to those for tensors.

- Consult Professor Bouman's slides for those generalizations.

- I should also mention that if you are befuddled by the partial derivatives with respect to matrices in Eqs. (13) and (15), check out the Wikipedia page on the topic of "Matrix Calculus".

Autograd's Computations of the Jacobians During the Forward Pass

- The overall implication of the chaining of the partial derivatives is fundamental to how Autograd does its thing: As the training data is pushed through a network, Autograd uses the finite-difference method to estimate all the partial derivatives that are needed for the learnable parameters in each layer of a network.

- The estimated partial derivatives are stored in a data-structure called **computational graph**.

- Subsequently, after the loss has been estimated at the final output, it is backpropagated through the network, layer to layer, and the parameter values updated using the previously computed partial derivatives during the forward pass.

- Starting with the next section, my goal is to help you develop deeper insights regarding the computational graph used by PyTorch.

Outline

1. Revisiting Gradient Descent 9
2. Problems with a Vanilla Application of GD 19
3. Stochastic Gradient Descent 21
4. **The Computational Graph Primer (CGP) 32**
5. CGP Demo 1: `graph_based_dataflow.py` 39
6. CGP Demo 2: `one_neuron_classifier.py` 52
7. CGP Demo 3: `multi_neuron_classifier.py` 63
8. CGP Demo 4: `verify_with_torchnn.py` 83
9. Autograd 90
10. Extending Autograd 94
11. Step Size Optimization for SGD 106

CGP: The Computational Graph Primer

- In order to convey a more computational understanding of what was described in the previous section, I have created a Python module called `ComputationalGraphPrimer` that you can download from:

  https://pypi.org/project/ComputationalGraphPrimer/

- CGP has the following educational goals:
  - To create simple handcrafted networks and to show the data structures that store the topology of the DAGs thus created.
  - To show how data can flow forward in a simple handcrafted DAG while calculating the partials of the output at a node with respect to the learnable parameters on the links converging at that node.
  - To consider specializations of the DAGs to handcrafted elementary neural networks for demonstrating the estimation of the gradients of the output in each layer with respect to the learnable parameters in that layer and to then show how to use the previously computed partial derivatives during the backpropagation of loss for updating the learnable parameters.
  - To verify the occurrence of learning in such handcrafted networks by comparing their loss vs. iterations performance with similar networks constructed with `torch.nn`.

The Computational Graph Primer (CGP) (contd.)

Your best entry into CGP is through the following Python scripts in the **Examples** directory of the distribution:

1. `graph_based_dataflow.py` Slide 39 – 50

   Demonstrates forward and backward data flows in a DAG while also calculating the partial derivatives needed for parameter update.

2. `one_neuron_classifier.py` Slide 52 – 62

   Uses the simplest possible handcrafted neural network to demonstrate the computation of the partial derivatives during forward propagation of data and using those derivatives for updating the learnable parameters during backpropagation.

3. `multi_neuron_classifier.py` Slide 63 – 82

   This is a generalization of the previous script to a multi-layer neural network. Both this and the previous networks use the Sigmoid activation function.

4. `verify_with_torchnn.py` Slide 83 – 89

   This script is for verifying the performance of the previous two scripts.

The Computational Graph Primer (CGP) (contd.)

- For the first three scripts on the previous slide, a network of nodes is created directly by the constructor of the CGP class. Subsequently, you execute a method whose names begins with `run_training_loop_ ...` for pushing the training data through the network.

- The last script, `verify_with_torchnn.py` is meant to verify that the handcrafted networks used in `one_neuron_classifier.py` and `multi_neuron_classifier.py` are indeed capable of learning as evidenced by the decreasing value of loss with the training iterations.

- The `run_training_loop_ ...` method called by `verify_with_torchnn.py` includes definitions for the `torch.nn` based classes for the one-neuron and multi-neuron models for creating those neural networks.

- As to what I mean by verification that is carried out by `verify_with_torchnn.py`, see the next slide.

The Computational Graph Primer (CGP) (contd.)

- The verification by verify_with_torchnn.py is only at the level of demonstrating that the networks are capable of learning. In terms of the performance — the rate at which the loss decreases with iterations — we can certainly expect the the verify_with_torchnn.py network to significantly outperform the handcrafted networks.

- In the slides that follow, I'll elaborate on each of the four scripts listed on Slide 34 and explain, with the help of the simple topologies of the networks, how, during the forward propagation of the data in a network, we can compute the partial derivatives that are subsequently used during backpropagation of the loss for updating the values of the learnable parameters.

- From a theoretical standpoint, what I'll be demonstrating through the four scripts will be the representational and computational aspects of the equations you saw in Slides 26 through 29.

The Computational Graph Primer (CGP) (contd.)

- I have distributed the overall explanation of the CGP module over the four scripts.

- For example, I have taken up the representational issues in my explanation of the `graph_based_dataflow.py`. **The representational issues relate to how we store the nodes and their interconnections for the forward and backward propagation of data.**

- Subsequently, through `one_neuron_classifier.py`, I have explained the basic mechanism in CGP for forward propagation of the data while you are computing the partial derivatives and for the backward propagation of loss as you update the learnable parameters. At this point, there is not much to the **chaining of the partial derivatives** that I mentioned on Slide 31 — because we have only one layer.

- The material on `one_neuron_classifier.py` also includes an explanation of how I generate the training data for the four scripts.

The Computational Graph Primer (CGP) (contd.)

- I have used `multi_neuron_classifier.py` to explain the fact that each expression in your list of expressions for configuring a neural network instantiates an object of type `Exp` that stores the name of the output node, the names of the input nodes, the names of the learnable parameters, etc.

- As you would expect, the main purpose of the `multi_neuron_classifier.py` is to explain how the chaining of the partial derivatives is carried out in the network.

- Again, all four scripts named on Slide 34 are in the **Examples** directory of the CGP module.

Outline

1. Revisiting Gradient Descent 9
2. Problems with a Vanilla Application of GD 19
3. Stochastic Gradient Descent 21
4. The Computational Graph Primer (CGP) 32
5. **CGP Demo 1: graph_based_dataflow.py 39**
6. CGP Demo 2: one_neuron_classifier.py 52
7. CGP Demo 3: multi_neuron_classifier.py 63
8. CGP Demo 4: verify_with_torchnn.py 83
9. Autograd 90
10. Extending Autograd 94
11. Step Size Optimization for SGD 106

Demo 1 Script: graph_based_dataflow.py

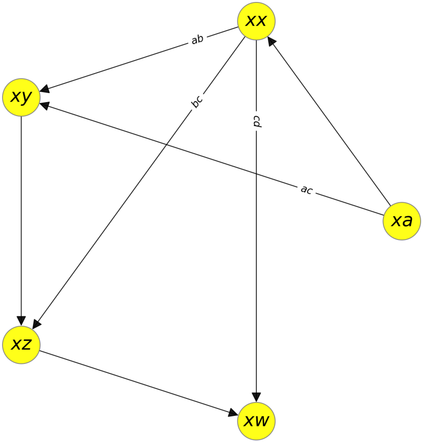
- You can create a DAG (Directed Acyclic Graph) with a statement like

  ```
  expressions = ['xx=xa^2',
                 'xy=ab*xx+ac*xa',
                 'xz=bc*xx+xy',
                 'xw=cd*xx+xz^3']
  ```

  where we assume that a symbolic name that starts with the letter x is a variable and that all other symbolic names are learnable parameters, and where we use the symbols $*$, $+$ and ^ for multiplication, addition, and exponentiation.

- The four expressions shown above contain five variables — xx, xa, xy, xz, and xw — and four learnable parameters: ab, ac, bc, and cd. The DAG that is generated by these expressions looks like what is shown on the next slide.

graph_based_dataflow.py (contd.)



In this handcrafted DAG (generated using CGP), xa is the only independent node (the same as the input node) and xw the only output node. The arcs have learnable parameters associated with them. Is it possible to estimate the learnable parameters? Certainly not without nonlinearities in the network.

graph_based_dataflow.py (contd.)

- **In the DAG shown on the previous slide, the variable xa is the only independent variable since it has no incoming arcs, and xw is the only output variable since it has no outgoing arcs.**

- A DAG of the sort shown on the previous slide is represented in CGP by two dictionaries: depends_on and leads_to. Here is what the depends_on dictionary would look like:

```
depends_on['xx']  =  ['xa']
depends_on['xy']  =  ['xa', 'xx']
depends_on['xz']  =  ['xx', 'xy']
depends_on['xw']  =  ['xx', 'xz']
```

- Something like depends_on['xx'] = ['xa'] is best read as "node xx depends on node xa." Similarly, the line depends_on['xz'] = ['xx','xy'] is best read aloud as "node xz depends on the nodes xx and xy." And so on.

graph_based_dataflow.py (contd.)

- Whereas the `depends_on` dictionary is a complete description of a DAG, for programming convenience, CGP also maintains another representation for the same graph, as provided by the `leads_to` dictionary. This dictionary for the same graph would be:

```
leads_to['xa']    =  ['xx', 'xy']
leads_to['xx']    =  ['xy', 'xz', 'xw']
leads_to['xy']    =  ['xz']
leads_to['xz']    =  ['xw']
```

- The `leads_to[xa] = [xx]` is best read as "the outgoing edge at node `xa` leads to node `xx`." Along the same lines, the `leads_to['xx'] = ['xy','xz','xw']` is best read as "the outgoing edges at node `xx` lead to the nodes `xy`, `xz`, and `xw`."

graph_based_dataflow.py (contd.)

- You create in instance of CGP by calling its constructor:

```
cgp = ComputationalGraphPrimer(
               expressions = ['xx=xa^2',
                              'xy=ab*xx+ac*xa',
                              'xz=bc*xx+xy',
                              'xw=cd*xx+xz^3'],
               output_vars = ['xw'],
               dataset_size = 10000,
               learning_rate = 1e-6,
               display_vals_how_often = 1000,
               grad_delta = 1e-4,
      )
```

  Subsequently, you invoke methods on the CGP instance to demonstrate how the data flows forward in the graph **while the node-to-node partial derivatives are calculated at the same time.**

- In the call shown above, we have designated the output var as `xw`. If you leave this option out, the module can figure out on its own as to which variables to designate as the output vars on the basis of there being no outgoing arcs at those vertices.

graph_based_dataflow.py (contd.)

- After you have constructed the instance cgp as shown on the previous slide, you would first need to call the parser as shown below in order to construct the computational graph:

  ```
  cgp.parse_expressions()
  ```

  By analyzing the topology of the graph, the parser figures out all the input vars and the output vars. If it sees that you have specified the output variables, it would only use those for the output.

- Next, you must generate the **ground-truth data** by a call like

  ```
  cgp.gen_gt_dataset(vals_for_learnable_params = {'ab':1.0, 'bc':2.0, 'cd':3.0, 'ac':4.0})
  ```

  This call gives us a mapping from randomly-generated values at the input nodes to the values produced at the output nodes when the learnable parameters are set as shown above. **Subsequently, we retain the input-output mapping as the ground-truth (GT) and forget the values used for the learnable parameters. The goal of learning would then be to recreate the values used for the learnable parameters.**

graph_based_dataflow.py (contd.)

As mentioned on the previous slide, you pretend that you don't know the true values used for the learnable parameters. So you call on the following method:

```
    cgp.train_on_all_data()
```

This method starts with randomly guessed values for the learnable parameters and goes through the following steps for each training sample in the GT dataset:

1. pushes each input value in the GT dataset through the network;
2. as the data propagates forward though the network, it simultaneously calculates the node-to-node partial derivatives using the finite difference method;
3. when the input training sample reaches the output nodes, it estimates the loss with respect to the GT output for the input; and, finally,
4. it backpropagates the loss and, using the partial derivatives computed during the forward propagation of the data, it updates the values of the learnable parameters.

graph_based_dataflow.py (contd.)

Each training iteration in the function train_on_all_data() called on the previous slide generates the following sort of output **at multiples of 1000 iterations** in the training loop:

```
Initial values for all learnable parameters: {'cd': 0.8134267156709766, 'ab': 0.6303253744046711, 'ac': 0.6490336075539102, 'bc': 0.0

======================  [Forward Propagation] Training with sample indexed: 1000 =======================
input values for independent variables:  {'xa': 0.8933787891895923}
predicted value at the output nodes:  {'xw': 2.1718835784922437}
loss for training sample indexed 1000: 212.7733678538975
estimated partial derivatives of vars wrt learnable parameters:
k=xx     v={'xx': {'cd': None, 'ab': None, 'ac': None, 'bc': None}, 'xa': {'cd': None, 'ab': None, 'ac': None, 'bc': None}, 'xz': {'c
k=xa     v={'xx': {'cd': None, 'ab': None, 'ac': None, 'bc': None}, 'xa': {'cd': None, 'ab': None, 'ac': None, 'bc': None}, 'xz': {'c
k=xz     v={'xx': {'cd': None, 'ab': None, 'ac': None, 'bc': 0.7981256609738974}, 'xa': {'cd': None, 'ab': None, 'ac': None, 'bc': No
k=xw     v={'xx': {'cd': 0.7981256609745913, 'ab': None, 'ac': None, 'bc': None}, 'xa': {'cd': None, 'ab': None, 'ac': None, 'bc': No
k=xy     v={'xx': {'cd': None, 'ab': 0.798125660973481, 'ac': None, 'bc': None}, 'xa': {'cd': None, 'ab': None, 'ac': 0.8933787891896

estimated partial derivatives of vars wrt other vars:
k=xx     v={'xx': None, 'xa': 1.786857578379708, 'xz': None, 'xw': None, 'xy': None}
k=xa     v={'xx': None, 'xa': None, 'xz': None, 'xw': None, 'xy': None}
k=xz     v={'xx': 0.08453220669404904, 'xa': None, 'xz': None, 'xw': None, 'xy': 0.9999999999998899}
k=xw     v={'xx': 0.8134345602339721, 'xa': None, 'xz': 3.970949898415288, 'xw': None, 'xy': None}
k=xy     v={'xx': 0.6303669260676603, 'xa': 0.6490757788568668, 'xz': None, 'xw': None, 'xy': None}

[sample index: 1000]: input val: {'xa': 0.8933787891895923}    vals for learnable parameters: {'cd': 0.8134339110108809, 'ab': 0.6303


======================  [Forward Propagation] Training with sample indexed: 2000 =========================
input values for independent variables:  {'xa': -0.5827703346221946}
predicted value at the output nodes:  {'xw': -0.5157700093872639}
loss for training sample indexed 2000: 38.09662138083069
estimated partial derivatives of vars wrt learnable parameters:
k=xx     v={'xx': {'cd': None, 'ab': None, 'ac': None, 'bc': None}, 'xa': {'cd': None, 'ab': None, 'ac': None, 'bc': None}, 'xz': {'c
k=xa     v={'xx': {'cd': None, 'ab': None, 'ac': None, 'bc': None}, 'xa': {'cd': None, 'ab': None, 'ac': None, 'bc': None}, 'xz': {'c
```

graph_based_dataflow.py (contd.)

- What you see on the previous slide is the output every 1000^{th} iteration in the training loop. The basic format of the information that is presented is:

```
input values for independent variables
predicted value at the output variables
loss
estimated partial derivatives of vars wrt learnable parameters
estimated partial derivatives of vars wrt other vars
```

- The partial derivatives are stored as two dictionaries at each node of the DAG. One dictionary is for the derivative of the output variable vis-a-vis all other variables in the network. [Obviously, depending on the network topology, some of these partials will be undefined.] The other dictionary is for the partials of the output variable with respect to all the learnable parameters. [Again, some of these partials will also remain undefined depending on the network topology.]

graph_based_dataflow.py (contd.)

- Here are a couple of additional methods defined for the ComputationalGraphPrimer module that you will find useful:

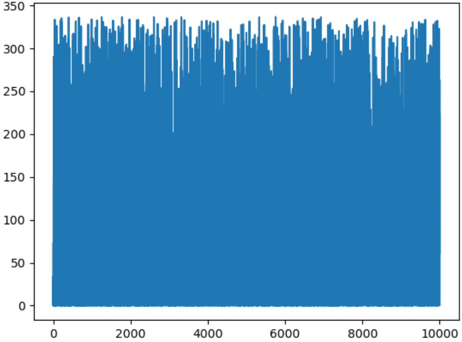
  ```
  cgp.display_network2()

  cgp.plot_loss()
  ```

  The first is for displaying the network graph and the second for plotting the loss as a function of the training iterations.

- Shown on the next slide is the loss as a function of iterations. Do not expect to see the loss decrease — simply because our DAG is not a learning network. For one, it does not include nonlinear activations at the nodes. My goal in this exercise was only to make you familiar with the notion of a DAG for forward and backward data flow.

graph_based_dataflow.py (contd.)



The fact that the loss is not decreasing is NOT surprising since there are no activation functions in our DAG. The goal of this exercise was merely to illustrate forward and backward dataflow in a DAG and how in principle you can calculate the partial derivatives during forward propagation for updating the parameters in backpropagation.

graph_based_dataflow.py (contd.)

- What about the details regarding the code that actually implements the forward propagation of data in the DAG, while we compute the needed partial derivatives? The following function in the CGP module shows the implementation code for the forward propagation of data in a general DAG while one also computes the partial derivatives:

  ```
  forward_propagate_one_input_sample_with_partial_deriv_calc()
  ```

- I am not going to go through the logic of the above function here because I have addressed that issue in considerable detail in the more interesting examples of the handcrafted one_neuron and multi_neuron cases of a general DAG that follow.

- The material presented for both the one_neuron and multi_neuron cases also includes the backpropagation of loss and the updating of the learnable parameters.

Outline

Demo 2 Script: one_neuron_classifier.py

- I'll now present what's got to be smallest possible example for illustrating forward propagation of the input data while computing the partial derivatives of the output with respect to the learnable parameters and then updating the parameters during backprop.
- A good starting point for understanding the demo based on the one-neuron model is the data that I use for this demo. As shown below, I generate random data for two different Gaussian distributions, with different means and variances. **The data generator is happy to create the multivariate data of any arbitrary dimensionality.**

```
def gen_training_data(self):
    num_input_vars = len(self.independent_vars)
    training_data_class_0 = []
    training_data_class_1 = []
    for i in range(self.dataset_size //2):
        for_class_0 = np.random.standard_normal( num_input_vars )
        for_class_1 = np.random.standard_normal( num_input_vars )
        for_class_0 = for_class_0 + 2.0                ## change the mean of Class 0
        for_class_1 = for_class_1 * 2 + 4.0            ## change the mean and variance of Class 1
        training_data_class_0.append( for_class_0 )
        training_data_class_1.append( for_class_1 )
    self.training_data = {0 : training_data_class_0, 1 : training_data_class_1}
```

one_neuron_classifier.py (contd.)

- Here is the data loader that for the demonstration. Note the implementation of the _getitem() method: it randomly chooses either a sample of Class 0 or a sample of Class 1. Also pay attention to data scaling and batch construction in the getbatch() method. The randomization used in creating a new batch means that there is no need to bring in the idea of epochs for training. [With epochs, you create a fresh randomization of the training data for each epoch.]

```
class DataLoader:

    def __init__(self, training_data, batch_size):
        self.training_data = training_data
        self.batch_size = batch_size
        self.class_0_samples = [(item, 0) for item in self.training_data[0]]   ## give label 0 to each class 0 sample
        self.class_1_samples = [(item, 1) for item in self.training_data[1]]   ## give label 1 to each class 1 sample

    def _getitem(self):
        cointoss = random.choice([0,1])                               ## returns either 0 or 1 for cointoss
        if cointoss == 0:
            return random.choice(self.class_0_samples)
        else:
            return random.choice(self.class_1_samples)
    def getbatch(self):
        batch_data,batch_labels = [],[]                               ## first list for samples, second for labels
        maxval = 0.0                                                  ## for approximate data normalization
        for _ in range(self.batch_size):
            item = self._getitem()
            if np.max(item[0]) > maxval:
                maxval = np.max(item[0])
            batch_data.append(item[0])
            batch_labels.append(item[1])
        batch_data = [item/maxval for item in batch_data]             ## normalize batch data
        batch = [batch_data, batch_labels]
        return batch
```

one_neuron_classifier.py (contd.)

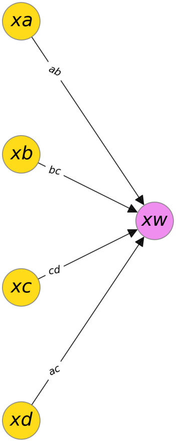
- Shown below is the call to the CGP constructor for constructing a one-neuron model.

- The neuron is constructing by parsing the expression that is supplied through the expressions constructor option. Since the node names start with the letter 'x', we have four input nodes and four corresponding learnable link weights in this example.

```
cgp = ComputationalGraphPrimer(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 1e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
      )
cgp.parse_expressions()
cgp.display_network2()
cgp.gen_training_data()
cgp.run_training_loop_one_neuron_model()
```

one_neuron_classifier.py (contd.)

- Shown below is the network constructed by the constructor call on the previous slide. The arc labels show the learnable parameters.



Using CGP, this is a handcrafted one-neuron network. The data aggregated at the output node is subject to Sigmoid activation.

one_neuron_classifier.py (contd.)

- Here is the training loop for the one-neuron model. Forward propagation is dispatched to the function forward_prop_one_neuron_model(). After that function has returned the prediction and the partial derivatives, we dispatch a call to the function backprop_and_update_params_one_neuron_mode().

```
def run_training_loop_one_neuron_model(self):
    training_data = self.training_data
    self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
    self.bias = random.uniform(0,1)                                              ## see comments in code for why we need this
    data_loader = DataLoader(training_data, batch_size=self.batch_size)
    loss_running_record = []
    i = 0
    avg_loss_over_iterations = 0.0                                    ##  Average the loss over iterations for printing out
                                                                       ##    every N iterations during the training loop.

    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples_in_batch = data[0]
        class_labels_in_batch = data[1]
        y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tuples_in_batch)     ##  FORWARD PROP of data
        loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2 \
                                          for i in range(len(class_labels_in_batch))])  ##  Find total loss over batch
        avg_loss_over_iterations += loss / float(len(class_labels_in_batch))
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_iterations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_iterations)
            print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))                 ## Display average loss
            avg_loss_over_iterations = 0.0                                                     ## Re-initialize avg loss
        y_errors_in_batch = list(map(operator.sub, class_labels_in_batch, y_preds))
        self.backprop_and_update_params_one_neuron_model(data_tuples_in_batch, y_preds, \
                                                  y_errors_in_batch, deriv_sigmoids)   ## BACKPROP loss
```

Forward Prop Through One-Neuron Classifier

- Here is the function for forward propagating the input data through the one-neuron classifier. Besides the prediction, the forward function need return only the partial derivatives of the Sigmoid at the operating points on the curves for each of the batch data tuples — **as justified on Slide 61**.
- In the code, exp_obj holds the info in the expression that defines the neural network.

```
def forward_prop_one_neuron_model(self, data_tuples_in_batch):       ## forward propagates one batch through NN
    output_vals, deriv_sigmoids = [], []
    for vals_for_input_vars in data_tuples_in_batch:
        input_vars = self.independent_vars                           ## [xa, xb, xc, xd] for example code
        vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))     ## sample vals for input vars
        exp_obj = self.exp_objects[0]          ## here there is only one expression object of type Exp [See Slide 61 for Exp]
        output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict, self.vals_for_learnable_params)
        output_val = output_val + self.bias
        output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))         ## apply sigmoid activation; see Eq. 22 on Slide 73
        deriv_sigmoid = output_val * (1.0 - output_val)              ## partial of sigmoid at output_val, see Eq. 23, Slide 73
        output_vals.append(output_val)                               ## output_vals for different input samples in batch
        deriv_sigmoids.append(deriv_sigmoid)                         ## collect sigmoid partials for samples in batch
    return output_vals, deriv_sigmoids                               ## see math on next slide
```

Backprop Through One-Neuron Classifier

- First note the parameter structure of the backprop function shown below. Each parameter is a list of what was either supplied to the forward propagation function or computed by the same for each training data tuple in the input batch.
- The batch averaging that is needed in accordance with Eq. (21) on Slide 61 is carried out for each learnable parameter separately in Line (A). The contribution from each batch data tuple is aggregated in the loop in Line (B).

```
def backprop_and_update_params_one_neuron_model(self, data_tuples_in_batch, predictions, y_errors_in_batch, deriv_sigmoids):
    input_vars = self.independent_vars
    input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]                  ## These two statements align the
    param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()}   ##   the input vars
    batch_size = len(predictions)
    vals_for_learnable_params = self.vals_for_learnable_params
    for i,param in enumerate(self.vals_for_learnable_params):                                    ## (A)
        ## For each param, sum the partials from every training data sample in batch
        partial_of_loss_wrt_param = 0.0
        for j in range(batch_size):                                                              ## (B)
            vals_for_input_vars_dict =  dict(zip(input_vars, list(data_tuples_in_batch[j])))
            partial_of_loss_wrt_param   +=   -  y_errors_in_batch[j] * \
                             vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
        partial_of_loss_wrt_param /=  float(batch_size)
        step = self.learning_rate * partial_of_loss_wrt_param
        ## Update the learnable parameters
        self.vals_for_learnable_params[param] += step
    y_error_avg = sum(y_errors_in_batch) / float(batch_size)
    deriv_sigmoid_avg = sum(deriv_sigmoids) / float(batch_size)
    self.bias += self.learning_rate * y_error_avg * deriv_sigmoid_avg    ## Update the bias
```

The Equations for Gradient Calculation

- The output of the neuron is given by the following formula in which $g()$ is the Sigmoid activation; N the number of input nodes; a_i the weight associated with the link from the i^{th} input node to the output node; b the bias; *each training sample* is given by $(x_1, x_2, \ldots, x_N)$; and where $\hat{y}$ is the output prediction:

$$\hat{y} \quad = \quad g\left(\sum_{i=1}^{N} x_i a_i + b\right) \tag{16}$$

- Therefore, we can write for the loss for each training sample where y^{gt} is the ground-truth label for the sample:

$$L \quad = \quad \left[y^{gt} \; - \; g\left(\sum_{i=1}^{N} x_i a_i + b\right)\right]^2 \tag{17}$$

- We need to take the derivatives of the loss with respect to the learnable parameters. Using $y_{err} = y^{gt} - \hat{y}$ for the prediction error and θ for the argument to the activation function, we have

$$\begin{aligned} \frac{\partial L}{\partial a_i} &= \; -2y_{err}\frac{\partial}{\partial a_i} g\left(\sum_{i=1}^{N} x_i a_i + b\right) \\ &= \; 2y_{err}\frac{\partial g}{\partial \theta}\frac{\partial}{\partial a_i}\left(\sum_{i=1}^{N} x_i a_i + b\right) \quad = \quad -2y_{err}\frac{\partial g}{\partial \theta} x_i \end{aligned} \tag{18}$$

Bringing in the Averaging Required by SGD

- SGD requires we average the loss over all the data samples in a batch before finding its partials with respect to the learnable parameters. Therefore, the equations shown on the previous slide need one more index for identifying the individual data samples in a batch. We will use the index j for that and assume that B is the batch size:

$$\hat{y}_j \;=\; g\left(\sum_{i=1}^{N} x_{i,j} a_i + b\right) \qquad j = 1, 2, \ldots, B \tag{19}$$

- The batch based estimated loss would now be given by

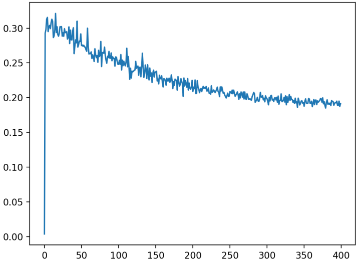
$$\begin{aligned} L &= \frac{1}{B}\sum_{j=1}^{B}\left[y_j^{gt} \;-\; \hat{y}_j\right]^2 \\ &= \frac{1}{B}\sum_{j=1}^{B}\left[y_j^{gt} \;-\; g\left(\sum_{i=1}^{N} x_{i,j} a_i + b\right)\right]^2 \end{aligned} \tag{20}$$

- Now we have the following expression for the partial derivative of L with respect to $a_i, i = 1, \ldots, N$ [See Eq. (26) on Slide 75 for what $\frac{\partial g}{\partial \theta}$ looks like]:

$$\frac{\partial L}{\partial a_i} \;=\; -\,\frac{2}{B}\sum_{j=1}^{B} y_j^{err}\,\frac{\partial}{\partial a_i}\, g\left(\sum_{i=1}^{N} x_i a_{i,j} + b\right) \;=\; -\,\frac{2}{B}\sum_{j=1}^{B} y_j^{err} \cdot \frac{\partial g}{\partial \theta} \cdot x_{i,j} \tag{21}$$

Loss vs. Training Iterations for the One Neuron Classifier

Shown below is how the loss decreases with the training iterations for the on-neuron model with Sigmoid activation. The plot shown is for 40,000 training iterations.



(a) $lr = 1e-3$

Loss vs. training-iterations

Outline

Demo 3 Script: multi_neuron_classifier.py

- In Demo 3, I will generalize what was demonstrated in Demo 2 to a multi-layer neural network.

- **I will continue to use the same data source and the same data loader.**

- The constructor call shown below creates a 3-layer network. That is, we now have an input layer, a hidden layer, and an output layer:

```
cgp = ComputationalGraphPrimer(
               num_layers = 3,
               ##  4 nodes in the input layer, 2 in the hidden layer, and 1 in the ouput layer:
               layers_config = [4,2,1],
               ##  input layer nodes: [xp, xq, xr, xs]   Hidden layer: [xw, xz]  Output layer: [xo]
               expressions = [’xw=ap*xp+aq*xq+ar*xr+as*xs’,
                              ’xz=bp*xp+bq*xq+br*xr+bs*xs’,
                              ’xo=cp*xw+cq*xz’],
               output_vars = [’xo’],
               dataset_size = 5000,
               learning_rate = 1e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )
```

multi_neuron_classifier.py (contd.)

- As you will see in the script multi_neuron_classifier.py in the Examples directory, the first call you invoke on the instance of the Primer class constructed above is: cgp.parse_multi_layer_expressions()

- If you peer inside the code for the method parse_multi_layer_expressions(), you will notice that each expression that you supply through the constructor option expressions is stored as instance of the Exp class with calls like

```
    exp_obj = Exp( exp, right, left, right_vars, right_params )
```

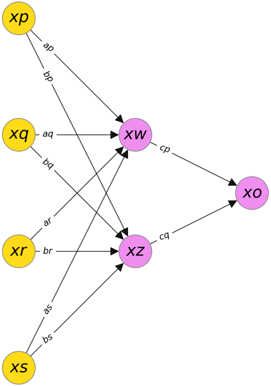
- The class Exp used above is defined as shown below:

```
    class Exp:
        def __init__(self, exp, body, dependent_var, right_vars, right_params):
            self.exp = exp
            self.body = body
            self.dependent_var = dependent_var
            self.right_vars = right_vars
            self.right_params = right_params
```

- The values for the variables exp, right, left, right_vars, and right_params are extracted from the individual expressions as you would expect.

multi_neuron_classifier.py (contd.)

- Shown below is the network created by the constructor call shown on Slide 64. **The learnable parameters are shown as arc labels.** Sigmoid activation is used for the data aggregated at each node.



Using CGP, this is a handcrafted 3-layer feedforward neural network for the multi-neuron example.

multi_neuron_classifier.py (contd.)

- Here is the training loop for the multi-neuron model. Note the call to the forward propagation function in line (A). The predictions and the derivatives calculated during forward propagation are communicated directly to the back propagation function called in line (D) through the instance variable assignments in lines (E) and (F) on Slide 69.

```
def run_training_loop_multi_neuron_model(self):
    training_data = self.training_data
    self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
    self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]      ## see comments in code for why we need this

    data_loader = DataLoader(self.training_data, batch_size=self.batch_size)
    loss_running_record = []
    i = 0
    avg_loss_over_literations = 0.0
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples = data[0]
        class_labels = data[1]
        self.forward_prop_multi_neuron_model(data_tuples)            ## FORW PROP works by side-effect              ## (A)
        predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers-1]   ## predictions
        y_preds =  [item for sublist in  predicted_labels_for_batch  for item in sublist]  ## get vals for predictions
        loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ## calc loss for batch
        loss_avg = loss / float(len(class_labels))
        avg_loss_over_literations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_literations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_literations)
            print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_literations))
            avg_loss_over_literations = 0.0
        y_errors = list(map(operator.sub, class_labels, y_preds))                                                   ## (B)
        y_error_avg = sum(y_errors) / float(len(class_labels))                                                      ## (C)
        self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)                               ## (D)
    plt.figure()
    plt.plot(loss_running_record)
    plt.show()
```

multi_neuron_classifier.py (contd.)

- Shown on the next slide is the forward function for the multi-neuron case. In addition to the predictions for the samples in a batch, the forward function must also calculate the the partial derivatives needed during the backprop step.

- In the code shown on the next slide, the expressions to evaluate for computing the pre-activation values at a node are stored at the layer in which the nodes reside. That is, the dictionary look-up self.layer_exp_objects[layer_index] returns the Expression objects for which the left-side dependent variable is in the layer pointed to layer_index. So the example shown above, self.layer_exp_objects[1] will return two Expression objects, one for each of the two nodes in the second layer of the network (that is, the layer indexed 1).

- The pre-activation values obtained by evaluating the expressions at each node are then subject to Sigmoid activation, followed by the calculation of the partial derivative of the output of the Sigmoid function with respect to its input.

multi_neuron_classifier.py (contd.)

- In the forward, the values calculated for the nodes in each layer are stored in the dictionary self.forw_prop_vals_at_layers[layer_index] and the gradients values calculated at the same nodes in the dictionary: self.gradient_vals_for_layers[layer_index].

```
def forward_prop_multi_neuron_model(self, data_tuples_in_batch):
    self.forw_prop_vals_at_layers = {i : [] for i in range(self.num_layers)}      ## stores the forw propagated vals in all layer
    self.gradient_vals_for_layers = {i : [] for i in range(1, self.num_layers)}   ## stores sigmoid gradients for all layers exce
    for vals_for_input_vars in data_tuples_in_batch:
        self.forw_prop_vals_at_layers[0].append(vals_for_input_vars)
        for layer_index in range(1, self.num_layers):
            input_vars = self.layer_vars[layer_index-1]
            if layer_index == 1:
                vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
            output_vals_arr = []        ##  This stores the outputs in the current layer
            gradients_val_arr = []      ##  This stores the gradients of Sigmoid in the current layer
            ##  In the following loop for forward propagation calculations, exp_obj is the Exp object that
            ##  is created for each user-supplied expression that specifies the network.  See the definition
            ##  of the class Exp (for 'Expression') by searching for "class Exp". It is also shown on Slide 65.
            for k,exp_obj in enumerate(self.layer_exp_objects[layer_index]):
                output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict, self.vals_for_learnable_params, input_
                output_val = output_val + self.bias[layer_index][k]
                ## Apply sigmoid activation:
                output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
                output_vals_arr.append(output_val)
                ## Calculate the partial of the activation function Sigmoid as a function of its input:
                deriv_sigmoid = output_val * (1.0 - output_val)
                gradients_val_arr.append(deriv_sigmoid)
                vals_for_input_vars_dict[ exp_obj.dependent_var ] = output_val
            ##  You have one "output_vals_arr" for each training data sample in batch:
            self.forw_prop_vals_at_layers[layer_index].append(output_vals_arr)
            ##  See the bullets in red on Slides 70 and 72 of my Week 3 slides for what needs
            ##  to be stored during the forward propagation of data through the network:
            ##  IMPORTANT: You have one "gradients_val_arr" for each data sample in batch.
            self.gradient_vals_for_layers[layer_index].append(gradients_val_arr)
```

multi_neuron_classifier.py (contd.)

- In the next few slides, I'll now explain the calculations that are carried out in the forward-propagation function on the previous slide.
- For the multi-neuron network in question, which is an example of a feedforward network, **the gradients that need to be remembered during the forward propagation of the data are the partial derivatives of the output of the activation function with respect to its input.**
- Recall we have a neural network with one input layer, one hidden layer, and one output later. Let w_1 be the matrix of link weights between the input and the hidden layer and w_2 the matrix of the link weights between the hidden layer and the output. The elements of the matrices w_1 and w_2 are our learnable parameters.
- Let h_{pre} denote the output of the hidden layer before the activation and h_{post} the output after the activation. We assume that the activation function is Sigmoid.

multi_neuron_classifier.py (contd.)

- In addition to the notation on the previous slide, we will use g_1 for the activation in the hidden layer and g_2 for the activation applied to the final output.

- Finally, let y_{pred} denote the predicted output at the final layer, and y_{gt} the ground-truth for a given input x. We can write for the loss

$$L \;=\; \|y_{gt} - y_{pred}\|^2 \tag{22}$$

- Recall that h_{post} is the output of the hidden layer post the activation. Also note that I am using the "column vector" representation for a vector in a matrix-vector product.

- For the partial of L with respect to the matrix w_2, we can write:

$$\begin{aligned} \frac{\partial L}{\partial w_2} &= -2y_{err} \cdot \left[\frac{\partial g_2}{\partial \theta} \cdot h_{post}\right]^T \\ &= -2y_{err} \otimes \left(\frac{\partial g_2}{\partial \theta} \cdot h_{post}\right) \end{aligned} \tag{23}$$

where, as previously stated, $y_{err} = y_{gt} - y_{pred}$.

multi_neuron_classifier.py (contd.)

- At the bottom of the previous slide, the symbol $\otimes$ is again the outer product of the two argument vectors. And *theta* is the vector of operating points on the nonlinear curve g_2. The product in the parentheses is an element-by-element multiplication of two vectors.

- Based on the last equation on the previous slide, we can say that **during forward propagation, when we are passing through the activation in the final layer, we need to remember the values for $\frac{\partial g_2}{\partial \theta}$ at the vector θ of activation points.**

- And during the backpropagation of the loss, we would need to compute the outer product between the vector of the prediction errors and the vector of post-activation values in the hidden layer after they are multiplied by $\frac{\partial g_2}{\partial \theta}$.

- Let's now address the question of updating the first link matrix w_1. In principle, for this update we are going to have compute the partial derivatives $\frac{\partial L}{\partial w_1}$.

`multi_neuron_classifier.py` (contd.)

- Since w_1 is one layer removed from where the loss L is calculated, we must invoke the chain rule to calculate the gradients of L with respect to the link weights w_1:

$$
\begin{aligned}
\frac{\partial L}{\partial w_1} &= \left[\frac{\partial L}{\partial y_{pred}}\right]^T \cdot \frac{\partial y_{pred}}{\partial h_{post}} \\
&= \left[\frac{\partial L}{\partial y_{pred}}\right]^T \cdot \left[\frac{\partial g_2}{\partial \theta 2} \cdot w_2\right]^T \cdot \frac{\partial h_{post}}{\partial w_1} \\
&= -2y_{err}^T \cdot [w_2]^T \cdot \left[\frac{\partial g_2}{\partial \theta_2}\right]^T \cdot \frac{\partial h_{post}}{\partial w_1} \\
&= -2\left(\frac{\partial g_2}{\partial \theta_2} \cdot w_2 \cdot y_{err}\right)^T \cdot \frac{\partial h_{post}}{\partial w_1} \\
&= -2\left(\frac{\partial g_2}{\partial \theta_2} \cdot w_2 \cdot y_{err}\right)^T \cdot \frac{\partial g_1(w_1 \cdot x_{input})}{\partial w_1} \\
&= -2[y_{err,1}]^T \cdot \frac{\partial g_1}{\partial \theta_1} \cdot \frac{\partial (w_1 \cdot x_{input})}{\partial w_1} \\
&= -2y_{err,1} \otimes \left(\frac{\partial g_1}{\partial \theta_1} \cdot x_{input}\right) \qquad (24)
\end{aligned}
$$

- See the explanations on the next slide.

multi_neuron_classifier.py (contd.)

- For the purpose of the derivation shown on the previous slide, we assume that our data vectors are row vectors.

- In the equations shown, $y_{err,1}$ is the prediction error backproped to the post-activation point for the first layer and $\theta_1 = w_1 \cdot x_{input}$ is the vector of operating points on the activation function curve g_1.

- Note that the second equation follows from the fact that $y_{pred} = g_2(w_2 \cdot h_{post})$.

- In the 4^{th} equation, we pull $\left(\frac{\partial g_2}{\partial \theta_2} \cdot w_2 \cdot y_{err}\right)$ together because that is the output prediction error backpropagated to the post-activation point in the hidden layer. Just as θ_1 was the vector of the activation points on the g_1 curve, θ_2 is the vector of activation points on the g_2 curve.

- In the fifth equation on the right, $g_1()$ is the activation function for the hidden layer. In the same equation, I have used the fact that the post-activation $h_{post} = g_1(w_1 \cdot x_{input})$.

multi_neuron_classifier.py (contd.)

- Based on the derivation on Slide 73, we can now say that **during forward propagation we must also remember the partial derivatives $\frac{\partial g_1}{\partial \theta 1}$ at the vector of activation points in the nonlinear function g_2.**

- As for the activation function — the Sigmoid — if θ is its input, its output will lie in the interval $(0, 1)$ and is given by

$$g(\theta) = \frac{1}{1 + e^{-\theta}} \tag{25}$$

and its derivative is given by

$$\frac{\partial g(\theta)}{\partial \theta} = g(\theta)(1 - g(\theta)) \tag{26}$$

- If we had used what's known as the ReLU activation function for $g()$ on the previous slide, instead of h_{post} we would have had

$$h_{rel} = ReLU(w_1 \cdot x) \tag{27}$$

which is the same as saying that

$$h_{rel} = \begin{cases} w_1 \cdot x & \text{when } w_1 \cdot x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{28}$$

multi_neuron_classifier.py (contd.)

- **I am now ready to talk about the backpropagation function that is shown on Slide 80.**

- When you compare Eq. (23) for estimating the partial derivatives of the loss with respect to the parameters in the link matrix w_2 with the form shown in Eq. (24) for the case when the partial derivatives are calculated for the parameters in the earlier occurring w_1, you can see how the updating of the link weights and the backpropagation of the prediction errors **can be extended to a neural network of any arbitrary depth through the steps summarized on the next slide**.

multi_neuron_classifier.py (contd.)

- Here are the key backpropagation steps in an arbitrary feedforward neural network:
  1. At layer l of the network, if w_l is the link matrix between layer $l-1$ and layer l, and if $error_l$ is the prediction error backproped to the post-activation point for layer l:

     $$gradient\ of\ \ loss\ wrt\ w_l \quad = \quad (error_l.g') \otimes output_{l-1} \qquad (29)$$

     where $output_{l-1}$ is the output of the previous layer (and, therefore, the input to layer l) and $g'()$ is the partial derivative of the activation function;
  2. Now backprop the prediction error $error_l$ to the post-activation point in the previous layer by ["mm" stands for Matrix Multiplication]:

     $$error_{l-1} \quad = \quad error_l.mm(w_l) \qquad (30)$$
  3. With regard to g', if you are using the ReLU activation, all you have to do to just zero out those elements where the forward propagated values to the activation function are negative. That gives you the backpropagated error on the pre-activation side of the layer $l-1$.

multi_neuron_classifier.py (contd.)

- Shown on Slide 80 is the backprop function. The following statements in lines (B) and (C) of the training loop code on Slide 67:

  ```
  y_errors = list(map(operator.sub, class_labels, y_preds))
  y_error_avg = sum(y_errors) / float(len(class_labels))
  ```

  mean that we send to the backprop function the batch averaged values for the prediction errors. The batch averaged values for the partial derivatives of the Sigmoid activation are calculated in the backprop function.

- About the backprop code on Slide 80, note that loop index variable `back_layer_index` starts with the index of the last layer. For our 3-layer example shown in the constructor call earlier, the value of `back_layer_index` starts with a value of 2, its next and final value is 1.

- As previously mentioned, the value of the `y_error` parameter is the batch averaged values for the prediction errors for the training samples in a batch.

multi_neuron_classifier.py (contd.)

- For the code shown on the next slide, note that the outermost loop iterates over the batch-based output produced by the forward-prop function shown earlier. SGD requires that, for each training data sample in a batch, we backpropagate the prediction errors from the final output to the first layer of nodes. During this propagation, we compute the gradients of Loss with respect to the learnable parameters. During this batch-based processing, we store away the partial gradients calculated. And we do the same thing to the changes we expect to see in the bias parameter that is also learned.

- At the end of batch-based processing mentioned above, we average the values of the gradients, both for the learnable parameters and the bias. About the bias, you have this parameter at every node where you aggregate the forward moving data.

- Finally, we use the averaged gradients for estimating the step to take for all the learnable parameters, including the bias.

multi_neuron_classifier.py (contd.)

Shown below is the backprop function for the multi-neuron case.

```
def backprop_and_update_params_multi_neuron_model(self, predictions, y_errors):
    ## Eq. (24) on Slide 73 of my Week 3 lecture says we need to store backproped errors in each layer leading up to the last:
    pred_err_backproped_at_layers =   [ {i : [None for j in range( self.layers_config[i] ) ]
                                                            for i in range(self.num_layers)} for _ in range(self.batch_size) ]
    ## This will store "\delta L / \delta w" you see at the LHS of the equations on Slide 73:
    partial_of_loss_wrt_params = {param : 0.0 for param in self.all_params}
    ## For estimating the changes to the bias to be made on the basis of the derivatives of the Sigmoids:
    bias_changes =   {i : [0.0 for j in range( self.layers_config[i] ) ]  for i in range(1, self.num_layers)}
    for b in range(self.batch_size):
        pred_err_backproped_at_layers[b][self.num_layers - 1] = [ y_errors[b] ]
        ## For the 3-layer network, the first val for back_layer_index is 2 for the 3rd layer:
        for back_layer_index in reversed(range(1,self.num_layers)):
            ## For the demo in Examples directory, "input_vals" is a list of 8 two-element lists
            ## since we have two nodes in the 2nd layer:
            input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
            ## This is a list eight one-element lists, one for each batch element that is calculated during forw prop:
            deriv_sigmoids =  self.gradient_vals_for_layers[back_layer_index]
            vars_in_layer  =  self.layer_vars[back_layer_index]                 ## A list like [’xo’]
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## A list like [’xw’, ’xz’]
            vals_for_input_vars_dict =  dict(zip(vars_in_next_layer_back, self.forw_prop_vals_at_layers[back_layer_index - 1][b]))
            ## For the next statement, note that layer_params are stored in a dict like
            ##       {1: [[’ap’, ’aq’, ’ar’, ’as’], [’bp’, ’bq’, ’br’, ’bs’]], 2: [[’cp’, ’cq’]]}
            ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
            layer_params = self.layer_params[back_layer_index]
            ## Creating a transpose of the link matrix, See Eq. 30 on Slide 77:
            transposed_layer_params = list(zip(*layer_params))
            for k,var1 in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    pred_err_backproped_at_layers[b][back_layer_index - 1][k] = \
                                  sum([self.vals_for_learnable_params[transposed_layer_params[k][i]]  \
                                  * pred_err_backproped_at_layers[b][back_layer_index][i] for i in range(len(vars_in_layer))])
            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]           ##  [’cp’, ’cq’]   for the end layer
                ## The following two statements align the    {’xw’: ’cp’, ’xz’: ’cq’}
                ##   and the input vars   {’cp’: ’xw’, ’cq’: ’xz’}
                input_vars_to_param_map = self.var_to_var_param[var]
                param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()}
```

(Continued on the next slide .....)

(...... continued from the previous slide)

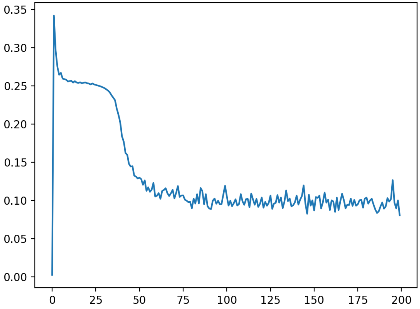
```
            ##  Update the partials of Loss wrt to the learnable parameters between the current layer
            ##  and the previous layer. You are accumulating these partials over the different training
            ##  data samples in the batch being processed.  For each training data sample, the formula
            ##  being used is shown in Eq. (29) on Slide 77 of my Week 3 slides:
            for i,param in enumerate(layer_params):
                partial_of_loss_wrt_params[param]   +=   pred_err_backproped_at_layers[b][back_layer_index][j] * \
                                                 vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[b][j]

            ##  data samples in the batch being processed.  For each training data sample, the formula
            ##  being used is shown in Eq. (29) on Slide 77 of my Week 3 slides:
            for i,param in enumerate(layer_params):
                partial_of_loss_wrt_params[param]   +=   pred_err_backproped_at_layers[b][back_layer_index][j] * \
                                                 vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[b][j]
        ##  We will now estimate the change in the bias that needs to be made at each node in the previous layer
        ##  from the derivatives the sigmoid at the nodes in the current layer and the prediction error as
        ##  backproped to the previous layer nodes:
        for k,var1 in enumerate(vars_in_next_layer_back):
            for j,var2 in enumerate(vars_in_layer):
                if back_layer_index-1 > 0:
                    bias_changes[back_layer_index-1][k] += \
                                          pred_err_backproped_at_layers[b][back_layer_index - 1][k] * deriv_sigmoids[b][j]

## Now update the learnable parameters.  The loop shown below carries out SGD mandated averaging
for param in partial_of_loss_wrt_params:
    partial_of_loss_wrt_param = partial_of_loss_wrt_params[param] /  float(self.batch_size)
    step = self.learning_rate * partial_of_loss_wrt_param
    self.vals_for_learnable_params[param] += step
##  Finally we update the biases at all the nodes that aggregate data:
for layer_index in range(1,self.num_layers):
    for k in range(self.layers_config[layer_index]):
        self.bias[layer_index][k]  +=  self.learning_rate * ( bias_changes[layer_index][k] / float(self.batch_size) )
```

multi_neuron_classifier.py (contd.)

Shown below is how the loss decreases with the training iterations for the multi-neuron model with Sigmoid activations. Compare this to the one-neuron case on Slide 62. In addition to the plunging drop in loss, note also that this output is for 20,000 iterations, which is half the number of iterations for the one-neuron case.



(a) $lr = 9e-2$

Outline

1 Revisiting Gradient Descent 9

2 Problems with a Vanilla Application of GD 19

3 Stochastic Gradient Descent 21

4 The Computational Graph Primer (CGP) 32

5 CGP Demo 1: graph_based_dataflow.py 39

6 CGP Demo 2: one_neuron_classifier.py 52

7 CGP Demo 3: multi_neuron_classifier.py 63

8 **CGP Demo 4: verify_with_torchnn.py 83**

9 Autograd 90

10 Extending Autograd 94

11 Step Size Optimization for SGD 106

Demo 4 Script: `verify_with_torchnn.py`

- In Demo 4, my goal is to compare the performance of the scripts

```
one_neuron_classifier.py
multi_neuron_classifier.py
```

  with similar networks constructed using components from PyTorch's `torch.nn` module.

- There are two reasons for this comparison:

  1. If the `torch.nn` based code says that it is possible to learn from a one-neuron network implementation, we would want the same from the script `one_neuron_classifier.py`. And the same would go for the three-layer "4-2-1" neural network in `multi_neuron_classifier.py`.
  2. An even more important reason for the comparison is for you to be impressed by the performance improvement you can get through step-size optimization in parameter update formulas when using `torch.nn`.

verify_with_torchnn.py (contd.)

- Shown on the next slide is the constructor call to the CGP class in the script `verify_with_torchnn.py`

- The lines following the constructor call that are labeled (A) and (B) determine whether your `torch.nn` based network will use a single neuron as in the `one_neuron_classifier.py` demo, or whether it will use multiple neurons as in the script `multi_neuron_classifier.py`.

- When the option value is `one_neuron`, we use the class `OneNeuronNet` for the learning network and when the option is `multi_neuron` we use the class `MultiNeuronNet`. These two classes based on `torch.nn` will be shown later.

- IMPORTANT: the `expression` option used in the constructor call is used ONLY for telling the system what dimensional multivariate data to generate for training the networks. The expression supplied serves absolutely no other purpose.

verify_with_torchnn.py (contd.)

- Note also there I have shown three different values for the parameter learning_rate. When using the multi_neuron option in line (B), you will get the best results with the learning rate set to 10^{-6}.

- On the other hand, if you choose the call in line labeled (A), your best choices for the learning rate are either 10^{-3} or 5×10^{-2}.

```
cgp = ComputationalGraphPrimer(
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],   # Only used to determine the data dimensiona
               dataset_size = 5000,
               learning_rate = 1e-6,             # For the multi-neuron option below
#               learning_rate = 1e-3,             # For the one-neuron option below
#               learning_rate = 5 * 1e-2,         # Also for the one-neuron option below
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
      )

##  This call is needed for generating the training data:
cgp.parse_expressions()
cgp.gen_training_data()
# cgp.run_training_with_torchnn('one_neuron')                   ## (A)
cgp.run_training_with_torchnn('multi_neuron')                   ## (B)
```

verify_with_torchnn.py (contd.)

- Shown below are the two classes that are used for the torch.nn based verification of the performance of one_neuron_classifier.py and multi_neuron_classifier.py.

- These classes are called in training loop shown on the next slide in lines (D) and (E). The code shown below goes at line (C).

```
class OneNeuronNet(torch.nn.Module):
    """
    This class is used when the parameter ’option’ is set to ’one_neuron’ in the call to
    this training function.
    """
    def __init__(self, D_in, D_out):
        torch.nn.Module.__init__( self )
        self.linear = torch.nn.Linear(D_in, D_out)
        self.sigmoid = torch.nn.Sigmoid()
    def forward(self, x):
        h_out = self.linear(x)
        y_pred = self.sigmoid(h_out)
        return y_pred

class MultiNeuronNet(torch.nn.Module):
    """
    This class is used when the parameter ’option’ is set to ’multi_neuron’ in the call to
    this training function.
    """
    def __init__(self, D_in, H, D_out):
        torch.nn.Module.__init__( self )
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)
    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred
```

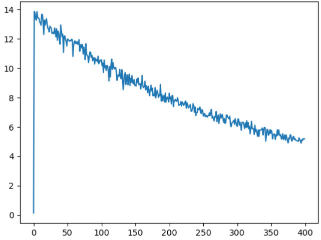
verify_with_torchnn.py (contd.)

- Here is the training loop for using the torch.nn based classes for the one-neuron and multi-neuron models.

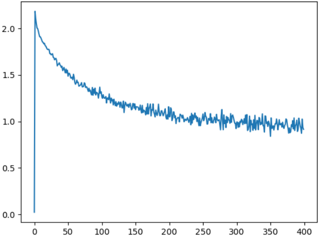
```
def run_training_with_torchnn(self, option):
    training_data = self.training_data
    ##  Code for the two class definition shown on the previous slide goes here                    ## (C)
    loss_running_record = []
    i = 0
    avg_loss_over_literations = 0.0
    if option == 'one_neuron':
        N,D_in,D_out = self.batch_size,self.input_size,self.output_size
        model = OneNeuronNet(D_in,D_out)                                                              ## (D)
    elif option == 'multi_neuron':
        N,D_in,H,D_out = self.batch_size,self.input_size,2,self.output_size
        model = MultiNeuronNet(D_in,H,D_out)                                                          ## (E)
    else:
        sys.exit("\n\nThe value of the parameter 'option' not recognized\n\n")
    criterion = torch.nn.MSELoss(reduction='sum')
    optimizer = torch.optim.SGD(model.parameters(), self.learning_rate)
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples = torch.FloatTensor( data[0] )
        class_labels = torch.FloatTensor( data[1] )
        # We need to convert the shape torch.Size([8]) into the shape torch.Size([8, 1]):
        class_labels = torch.unsqueeze(class_labels, 1)
        y_preds = model(data_tuples)
        loss = criterion(y_preds, class_labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        avg_loss_over_literations += loss
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_literations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_literations)
            print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_literations))
            avg_loss_over_literations = 0.0
    plt.figure()
    plt.plot(loss_running_record)
    plt.show()
```

verify_with_torchnn.py (contd.)

Shown below is how the loss decreases with the training iterations for the multi-neuron and the one-neuron cases when using torch.nn based networks:



(a) multi-neuron network



(b) one-neuron network

Outline

1. Revisiting Gradient Descent 9
2. Problems with a Vanilla Application of GD 19
3. Stochastic Gradient Descent 21
4. The Computational Graph Primer (CGP) 32
5. CGP Demo 1: `graph_based_dataflow.py` 39
6. CGP Demo 2: `one_neuron_classifier.py` 52
7. CGP Demo 3: `multi_neuron_classifier.py` 63
8. CGP Demo 4: `verify_with_torchnn.py` 83
9. **Autograd 90**
10. Extending Autograd 94
11. Step Size Optimization for SGD 106

Autograd for Automatic Calculation of Gradients

- I have already mentioned several things about Autograd in the material we have covered so far. In this section, I am now going to summarize how this module functions.
- With Autograd, during the forward pass of a training sample, a computational graph is constructed and the partial derivatives calculated in the same manner as in `ComputationalGraphPrimer` — although Autograd uses a more sophisticated computational graph in which the operators on the tensors define the nodes of the graph as opposed to the tensors themselves.
- It would not be too difficult to extend the implementation of the `ComputationalGraphPrimer` module so that the operators inside the expressions serve as the nodes of the graph. However, the basic demonstration of the principle of automatic differentiation during the forward pass of the training data would remain unchanged.

Autograd for Automatic Calculation of Gradients (contd.)

- After the loss is calculated at the output, the partial derivatives calculated during the forward pass are used to propagate the loss backwards and the gradient of the loss computed with respect to the parameters encountered. This is initiated by the following statement in your code:

```
    loss.backward()
```

- **For this behavior of Autograd, all you have to do is to set the `requires_grad` attribute of the tensors that define the learnable parameters to `True`. When you are using the automation provided by `torch.nn`, that module takes care of setting the `requires_grad` property true for all the learnable parameters.**

- The value of the gradient itself is stored in the attribute `grad` of tensors involved.

Static vs. Dynamic Computational Graphs

- PyTorch's computational graphs are dynamic, in the sense that a new graph is created in each forward pass.
- On the other hand, the computational graphs constructed by Tensorflow are static. That is, the same graph is used over and over in all iterations during training.
- In general, static graphs are more efficient because you need to optimize them only once. Optimization generally consists of distributing the computations over the graph nodes across multiple GPUs if more than one GPU is available or just fusing some nodes of the graph if the resulting logic won't be impacted by such fusion.
- Static graphs do not lend themselves well to recurrent neural computations because the graph itself can change from iteration to iteration.

Outline

1. Revisiting Gradient Descent 9
2. Problems with a Vanilla Application of GD 19
3. Stochastic Gradient Descent 21
4. The Computational Graph Primer (CGP) 32
5. CGP Demo 1: `graph_based_dataflow.py` 39
6. CGP Demo 2: `one_neuron_classifier.py` 52
7. CGP Demo 3: `multi_neuron_classifier.py` 63
8. CGP Demo 4: `verify_with_torchnn.py` 83
9. Autograd 90
10. **Extending Autograd 94**
11. Step Size Optimization for SGD 106

Extending Autograd

- Being highly object-oriented, you would think that PyTorch would give you all kinds of freedom in extending the platform. But that's not the case.
- On account of how PyTorch sits on top of the C++ code base that knows how to work the GPUs, the normal code extension facilities that one attributes to OO platforms do not apply to PyTorch.
- You are allowed to extend only two things in PyTorch: Autograd and the torch.nn module, and that too in only certain specific ways. In this section, I'll show what it is you have to do to extend Autograd.
- I'll illustrate how to extend Autograd with the help of the inner class `AutogradCustomization` in my `ComputationalGraphPrimer` module.

Extending Autograd (contd.)

- The Examples subdirectory of the CGP distribution contains a script named `extending_autograd.py` with the following code:

```
from ComputationalGraphPrimer import *
cgp = ComputationalGraphPrimer(                                                      ## (A)
                  learning_rate = 1e-3,
                  epochs = 5,
              )

ext_auto = ComputationalGraphPrimer.AutogradCustomization(                           ## (B)
                                                          cgp = cgp,
                                                          num_samples_per_class = 1000,
                                                        )
ext_auto.gen_training_data()                                                         ## (C)
ext_auto.train_with_straight_autograd()                                              ## (D)
ext_auto.train_with_extended_autograd()                                              ## (E)
```

where in line (A), we construct an instance of the `ComputationalGraphPrimer` class. Subsequently, in line (B), we construct an instance of the inner class `AutogradCustomization` that has the code for demonstrating how to extend PyTorch's Autograd module.

Extending Autograd (contd.)

- In line (C) in the previous slide, we generate the labeled training data from a 2D multivariate Gaussian. The definition of the function `ext_auto.gen_training_data()` invoked in line (C) has the following code:

```
mean1,mean2   = [3.0,3.0], [5.0,5.0]
covar1,covar2 = [[1.0,0.0], [0.0,1.0]], [[1.0,0.0], [0.0,1.0]]
data1 = [(list(x),1) for x in np.random.multivariate_normal(mean1, covar1,self.num_samples_per_class)]
data2 = [(list(x),2) for x in np.random.multivariate_normal(mean2, covar2,self.num_samples_per_class)]
training_data = data1 + data2
random.shuffle( training_data )
```

- As you see, we are drawing training samples from two bivariate Gaussian distributions, with different means but identical covariances.

- The next slide shows the definition of the function `train_with_straight_autograd()` that has been invoked in line (D) of the previous slide. By “straight autograd”, I mean Autograd without any modifications.

Extending Autograd (contd.)

- Before explaining how you can extend Autograd, in order to compare the before and after results, let's first look at the implementation of the function train_with_straight_autograd() that's based on Autograd as it comes:

```
def train_with_straight_autograd(self):
    dtype = torch.float                                                               # (1)
    D_in,H,D_out = 2,10,2                                                             # (2)
    w1 = torch.randn(D_in, H, device="cpu", dtype=dtype)                              # (3)
    w2 = torch.randn(H, D_out, device="cpu", dtype=dtype)                             # (4)
    w1 = w1.to(self.cgp.device)                                                       # (5)
    w2 = w2.to(self.cgp.device)                                                       # (6)
    w1.requires_grad_()     ## in-place setting of requires_grad attribute to True    # (7)
    w2.requires_grad_()     ## in-place setting of requires_grad attribute to True    # (8)
    Loss = []                                                                         # (9)
    for epoch in range(self.cgp.epochs):                                              # (10)
        for i,data in enumerate(self.training_data):                                  # (11)
            input, label = data                                                       # (12)
            x,y = torch.as_tensor(np.array(input)), torch.as_tensor(np.array(label))  # (13)
            x,y = x.float(), y.float()                                                # (14)
            if self.cgp.device:                                                       # (15)
                x,y = x.to(self.cgp.device), y.to(self.cgp.device)                    # (16)
            y_pred = x.view(1,-1).mm(w1).clamp(min=0).mm(w2)                          # (17)
```

(Continued on the next slide .....)

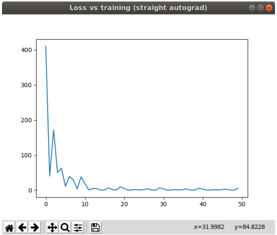
(...... continued from the previous slide)

```
loss = (y_pred - y).pow(2).sum()                                # (18)
loss.backward()                                                 # (19)
with torch.no_grad():                                           # (20)
    w1 -= self.cgp.learning_rate * w1.grad                      # (21)
    w2 -= self.cgp.learning_rate * w2.grad                      # (22)
    w1.grad.zero_()                                             # (23)
    w2.grad.zero_()                                             # (24)
```

Explanation of the code:

The statement in line (2) supplies the number of nodes to be used for a one-hidden-layer neural network. Line (3) defines the link matrix for the input to the hidden layer and line (4) for the hidden layer to the output. We intentionally create the learnable parameters for w1 and w2 in the cpu and not in the gpu in order for the results to be reproducibly the same regardless of which devices is being used for computing. Of course, for reproducibility, you will also have to set the different seeds to zero.

The rest of the code is self explanatory. In line (11), we get hold of the input/output data pairs from the training dataset created by the data generator on Slide 54. Subsequently, in line (17), we push the training sample through the network and calculate the predicted label for the input. A plot of the loss against the training iterations is shown below:



Extending Autograd (contd.)

- Now we are all set for me to explain how you can extend Autograd. The first thing you need to do is to define your verb class that can: (1) trap a training sample in its forward pass through the network; (2) modify the training sample if necessary; (3) use a context variable to remember whatever is deemed important about the change that was made to the training sample in its forward journey; and then (4) on the backward pass related to the same training sample, recall what was stored away in the context variable, and do whatever is needed.

- This class that your code must define must include implementations for two static methods `forward()` and `backward()`.

- To satisfy these requirements, the code for the inner class `AutogradCustomization` of the CGP module contains the class definition shown on the next slide. The name of this class is `DoSillyWithTensor`.

Extending Autograd (contd.)

- Shown below is the class DoSillyWithTensor mentioned on the previous slide:

```
class DoSillyWithTensor(torch.autograd.Function):                                   # (1)
    @staticmethod                                                                   # (2)
    def forward(ctx, input):                                                        # (3)
        input_orig = input.clone().double()                                         # (4)
        input = input.to(torch.uint8).double()                                      # (5)
        diff = input_orig.sub(input)                                                # (6)
        ctx.save_for_backward(diff)                                                 # (7)
        return input                                                                # (8)

    @staticmethod                                                                   # (9)
    def backward(ctx, grad_output):                                                 # (10)
        diff, = ctx.saved_tensors                                                   # (11)
        grad_input = grad_output.clone()                                            # (12)
        grad_input = grad_input + diff                                              # (13)
        return grad_input                                                           # (14)
```

Explanation of the code:

The parameter input in line (3) is set to the training sample that is being processed by an instance of DoSillyWithTensor in the forward() of the network. In line (4), we first make a deep copy of this tensor (which should be a 32-bit float) and then we subject the copy to a conversion to a one-byte integer in line (5). This should cause a significant loss of information in the training sample. In line (6), we calculate the difference between the original 32-bit float and the 8-bit version and store it away in the context variable ctx. Subsequently, we retrieve this quantization error during the backward pass in line (11) and add it to the value in the grad attribute of the tensor.

Extending Autograd (contd.)

- Finally, we are ready to talk about the call to the function train_with_extended_autograd() in line (E) on Slide 96. Here is how that function is implemented in the inner class AutogradCustomization of the CGP module:

```
def train_with_extended_autograd(self):
    dtype = torch.float
    D_in,H,D_out = 2,10,2
    w1 = torch.randn(D_in, H, device="cpu", dtype=dtype)
    w2 = torch.randn(H, D_out, device="cpu", dtype=dtype)
    w1 = w1.to(self.cgp.device)
    w2 = w2.to(self.cgp.device)
    w1.requires_grad_()
    w2.requires_grad_()
    Loss = []
    for epoch in range(self.cgp.epochs):
        for i,data in enumerate(self.training_data):
            do_silly = ComputationalGraphPrimer.AutogradCustomization.DoSillyWithTensor.apply  # (1)
            input, label = data
            x,y = torch.as_tensor(np.array(input)), torch.as_tensor(np.array(label))
            x = do_silly(x)                                                                    # (2)
            x,y = x.float(), y.float()
            x,y = x.to(self.cgp.device), y.to(self.cgp.device)
```

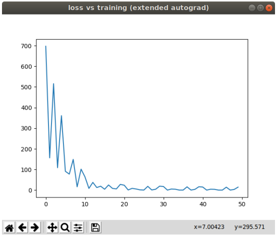
(Continued on the next slide .....)

(...... continued from the previous slide)

```
y_pred = x.view(1,-1).mm(w1).clamp(min=0).mm(w2)
loss = (y_pred - y).pow(2).sum()
loss.backward()
with torch.no_grad():
    w1 -= self.cgp.learning_rate * w1.grad
    w2 -= self.cgp.learning_rate * w2.grad
    w1.grad.zero_()
    w2.grad.zero_()
```

Explanation of the code:

Except for the lines labeled (1) and (2), this code is exactly the same as what you saw earlier on Slides 98 and 99. The call in line (1) constructs an instance of DoSillyWithTensor. Note that this instance is callable. Subsequently, in line (2), the training sample is being processed by the do_silly instance in line (1). The loss of precision caused by the extension to Autograd is evident in the following plot of loss versus training iterations. Compare this result with the one shown earlier in Slide 99.



About the Code Examples Shown in This Section

- The code shown on Slides 96 through 103 represented examples of manually constructed neural networks (as opposed to networks using, say, Torch.nn). We ourselves defined the matrices $w1$ and $w2$ for the link weights, with $w1$ representing the learnable weights between the input layer and the hidden layer and $w2$ doing the same for the weights between the hidden layer and the output layer.

- When you are using PyTorch and you wish to specify a network manually, you must also identify for PyTorch as to which the data entities constitute learnable weights. **You can only do that if your learnable entities are tensors and if you have set the `requires_grad` attributes of those tensors to True. That's what was done in lines (7) and (8) of the code shown on Slide 98.**

- Additionally, when you specify a network manually in PyTorch, you must also specify what it takes to calculate the predictions in the output layer — as illustrated in the first bullet on the next slide for the code on Slide 98.

About the Code Examples Shown (contd.)

- In the code shown on Slide 98, the following statement calculates the prediction at the output:

```
y_pred = x.view(1,-1).mm(w1).clamp(min=0).mm(w2)
```

- Do you understand what is being accomplished by the calls to `view()` and `mm()`? **These are very useful functions to get to know for your own attempts at code writing.**

[**EXPLANATION**: In the above example, the input data element `x` is 2-dimensional. And the hidden layer in the code on Slide 98 is 10-dimensional. So `w1` is 2 × 10. We can calculate the pre-activation data aggregation at the hidden layer either as ${w_1}^T x$ or as $x^T w_1$. Either one of these will return a 10-dimensional entity, the first as a column vector and the second as a row vector of the same dimensionality. The implementation shown above uses the latter. We call on `view()` to reshape x as a 1×2 matrix. The cool thing about calling `view()` with one of its argument set to -1 is that it lets PyTorch figure out on its own what that should be. In what it does, `torch.view()` is similar to what is accomplished by numpy's `reshape()` function. However, unlike numpy's `reshape()`, the new tensor returned by `torch.view()` shares the underlying data with the original tensor. In that sense, what `torch.view()` returns is just a new "view" of the old data. Said another way, numpy's `reshape()` allocates fresh memory for the reshaped data object. On the other hand, `tensor.view()` does not make a copy of anything.]

[**EXPLANATION**: `torch.mm()` does a regular matrix multiplication of two argument matrices. So if you declare `mat1 = torch.randn(2,4)` and `mat2 = torch.randn(4,6)`, the call `mat1.mm(mat2)` will return a 2x6 matrix.]

Outline

How the Shape of the Cost Function Surface Can Affect SGD

- Both GD and SGD run into an interesting problem if the otherwise well-behaved cost-function surface is a narrow valley as shown below, as opposed to a more rounded valley.

```
           _________________________________________________
          /     _______________________________________     \
         |     /     _______________________________    \    |
         |    |    /               X                \    |   |
         |    |    \________________________________/    |   |
         |     \_______________________________________ /    |
          \_________________________________________________/
             x
           start
            /
           /
direction of the steepest descent at the
initial point in the parameter hyperplane
```

shown are equi-cost
contours, with the
inner contours
corresponding to
lower cost

- Such a cost function is problematic because the direction of the steepest descent at the start point in the parameter hyperplane will point in a straight downhill direction along the shortest path to the bottom *spine* of the valley — this would **not** be the direction toward the optimum.

- What do you think would happen for the case shown above if we maintained the same learning rate all through the learning process?

How the Shape of the Cost-Function Surface Can Affect SGD (contd.)

- If the same learning rate is used for all the iterations, the solution path in this case is likely to look like what is shown below.

```
                       /\      /\      /\      /\
                      /  \    /  \    /  \    /  V
The riverbed=>       /    \  /    \  /    \  /          ----->  toward the goal
                    /      \/      \/      \/
                   /
                  /
                start
```

- After getting to the bottom spine of the valley, the solution path will crisscross the riverbed, although, overall, the direction of movement would be toward the optimum point because the negatives of all locally computed gradients will have a slight slant toward the point where the cost function is the absolute minimum.

- The problem with such a solution path is that you may never get a good sense of whether you are sufficiently close to the goal.

Why We Need Momentum

- On the other hand, if we could decrease the learning rate with the iterations, you are likely to see the following sort of a solution path:

```
                       /\
The riverbed=>        /  \    /\  /\  ___  >  toward the goal
                     /    \  /  \/  \/
                    /      \/
                   /
                  /
                start
```

- With or without any decrease in the learning rate, the tendency of a solution path to oscillate around can be significantly arrested by using what is now referred to as the **momentum**.

- In addition to dampening the oscillations, using momentum in step-size calculations also significantly improves the convergence of a gradient descent algorithm.

- To explain the idea of momentum, on the next slide I'll go back to our basic formula for calculating the next value for a parameter in the parameter hyperplane from the current value of the gradient of the loss with respect to the parameter in question.

The Idea of Momentum

- Here's going back to our basic equation for extending the solution path (see Slide 16):

$$\begin{aligned} \mathbf{p}_{k+1} &= \mathbf{p}_k - 2 \cdot \alpha \cdot J_{\mathbf{F}}(\mathbf{p}_k) \cdot \epsilon_k \\ &= \mathbf{p}_k - \delta p_k \end{aligned} \tag{31}$$

  where $\delta p_k = 2 \cdot \alpha \cdot J_{\mathbf{F}}(\mathbf{p}_k) \cdot \epsilon_k$ represents the change in the solution path at step k. As you know already, α is the learning rate.

- In Eq. (31) above, the basic idea of momentum is to compare the current value of δp_k with its previous value δp_{k-1}. If both are pointing in the same direction, that means you are continuing to go downhill and **so you might as well accelerate and increase the step size. However, should the two path increments point in different directions, you had better watch out.**

- From a computational standpoint, the above idea is implemented by computing the step updates separately at each iteration using a fraction of the previous update and only then modifying the current point on the solution path.

The Idea of Momentum (contd.)

- That is, at iteration k, let δp_k be the new momentum-modified value for the path increment. We first calculate δp_k as follows:

$$\delta p_k = \gamma \cdot \delta p_{k-1} \quad + \quad 2 \cdot \alpha \cdot J_{\mathbf{F}}(\mathbf{p}_k) \cdot \epsilon_k \tag{32}$$

  where γ, known as the momentum coefficient, is the fraction of the previous value of δp_{k-1} that we retain for the present value of δp_k. Note that the learning rate only multiplies the gradient of the loss at the current iteration on the solution path in the hyperplane defined by the parameter vector p.

- Subsequently, we extend the solution path by computing:

$$p_{k+1} \quad = \quad p_k \quad - \quad \delta p_k \tag{33}$$

[**To understand as to what is accomplished by the above, assume that the momentum term $\gamma = 1.0$. Also assume for a moment that the learning rate α is set to 1.0. Under these conditions, assuming the latest gradient is the same as the gradient computed at the previous time step, you would in effect be doubling the gradient at each time step. So as long as you stay on the same side of the mountain, you would accelerate exponentially down the hill until you get to the ravine at the bottom. At some point, these accelerating leaps down the mountain will cross to the other side of the ravine at the bottom in one giant leap. At this point, the sign of the second term in Eq. (32) will flip and now you will have not only a sudden reduction in the step size but also possibly a reversal in its direction back towards the ravine at the bottom of the valley.**]

Another Issue Related to Step-Size Optimization — Sparse Gradients

- While using momentum as explained so far considerably improves the convergence properties of any gradient descent based approach, it is not the last word in step size optimization.

- You see, in neural-network based learning (involving images for the sake of making a point) it is highly unlikely that any one image, or even a given batch of images, would be equally sensitive to all the learnable parameters in a network. [**For example, consider a system that is designed to recognize cars with street scene backgrounds, bicycles with trail backgrounds, trees with wooded area backgrounds, boats with bodies of water in the background, etc. Now assume that you are using different convolutional filters to extract different types of features from the images. Since the elements of the convolutional operators are in the parameter vector p, during the forward pass the output of each layer would be sensitive to only a small number of elements of the p vector. This would result in sparse Jacobians for the individual layers.**]

- This results in what's known as the phenomenon of **sparse gradients**.

- When you have sparse gradients, then for each SGD iteration, you will see zero partial derivatives in a large portion of the space spanned by the parameter vector p.

Step Size and Sparse Gradients (contd.)

- When you have sparse gradients, it does not make any sense to use the same learning rate α for all the dimensions of the parameter hyperplane.

- The first well-known step-size estimation algorithm that adapted the learning rate to the different parameters at each training step was AdaGrad (for Adaptive Gradients).

- AdaGrad is also based on the intuition that, should the departure from the zero derivatives be rare for some of the dimensions of the parameter space, one should take significantly larger steps for those dimensions whenever an opportunity presents itself — meaning whenever those partial derivatives become non-zero — **simply because the rareness of such occurrences could mean that those dimensions carry high class discriminatory information**.

AdaGrad's Answer to Sparse Gradients

- To see how AdaGrad adapts the updating process to each parameter separately, I'll use the following notation:

$$\begin{aligned} p_k &= \text{the full parameter vector at training iteration } k \\ p_{k_i} &= \text{the } i^{th} \text{ component of the parameter vector at iteration k} \\ g_k &= \nabla_p L(p_k) \;\; \text{this is the full gradient of the loss at iteration } k \\ g_{k_i} &= \text{the } i^{th} \text{ component of the vector } g_k \end{aligned} \tag{34}$$

- The updating formula used in regular SGD, when examined separately for each dimension of the parameter vector, can be expressed as:

$$p_{k+1_i} \quad = \quad p_{k_i} \quad - \quad \alpha \cdot g_{k_i} \tag{35}$$

- Now consider the following updating formula as a stepping stone to the final formula used in AdaGrad:

$$p_{k+1_i} \quad = \quad p_{k_i} \quad - \quad \frac{\alpha}{\sum_k (g_{k_i})^2} \cdot g_{k_i} \tag{36}$$

AdaGrad's Answer to Sparse Gradients (contd.)

- In the formula shown at the bottom of the previous slide, we sum the squares of all the previous values of the i^{th} component of the gradient.

- Should i^{th} component of the gradient take on any non-zero values rarely, the term in the denominator of the fraction would be relatively small. So, relatively speaking such a component would receive a greater weightage during the updating process in relation to the components that are frequently nonzero.

- Although, conceptually, it would seem that the above formula would do the job of the adaptation we need, in practice the formula could run into a division-by-zero kind of error if the gradient with respect to any dimension has yet to receive any non-zero value. Another problem with the above formula is that, in practice, it is seen that you get the best results if you use the square-root of the summation in the denominator. So here is a better formula:

$$p_{k+1_i} \quad = \quad p_{k_i} \quad - \quad \frac{\alpha}{\sqrt{\sum_k (g_{k_i})^2 + \epsilon}} \cdot g_{k_i} \tag{37}$$

RMSProp's Mod to the AdaGrad Formula

- While AdaGrad's update formula shown on the previous slide represented a considerable advance over using the same learning rate for all the parameter, it was seen to run into problems when the monotonically increasing value for the denominator caused the learning rate for a parameter to become vanishing small. When that happened, there was no further learning for that parameter.

- This problem was rectified by in a variant of the AdaGrad algorithm known as RMSprop by replacing the monotonically increasing summation in the denominator with its average over the training iterations.

- **The step-size optimization algorithm that is now used universally is known as Adam which stands for "Adaptive Moment Estimation".**

- Adam pulls together the momentum based logic presented earlier with the logic for dealing with sparse gradients into a single framework.

The Adam Optimizer

- Here is the basis for how Adam manages to pull into the same update framework what appear to be disparate considerations for optimizing the step sizes in the parameter hyperplane:
    - The notion of momentum boils down to keeping a running decaying average of the mean of the past gradient components — this would be the first moment of the gradient components.
    - And the need to be adaptive to the the different components of the gradient as in AdaGrad / RMSprop requires that we keep running tabs on the second moment of the gradient component values.
- Adam estimates on a running-average basis the first moment, denoted m_k below, and the second moment, denoted v_k, using the following formulas:

$$\begin{aligned} m_k &= \beta_1 m_{k-1} + (1-\beta_1) g_k \\ v_k &= \beta_2 . v_{k-1} + (1-\beta_2)(g_k)^2 \end{aligned} \tag{38}$$

  Recall that, as defined in Eq. (34), g_k is the gradient of the loss L at iteration k.

The Adam Optimizer (contd.)

- The recursions are initialized by setting $m_0 = v_0 = 0$. And typical values for the constants β_1 and β_2 are close to 1.0. **[It is good to commit to memory that the user-supplied β_1 is meant for the first moment of the gradients — the same thing as the momentum. And that the user-supplied β_2 is meant for the second moment values that are needed for adaptation to the parameters with respect to which the gradients are sparse. Obviously, another user supplied value for using Adam is ϵ to deal with the problem of division by zero as you saw in the formula shown for AdaGrad.]**

- There is, however, a price to pay for the zero initialization for m_0 and v_0, it biases the values of the two moments toward zero for several training iterations. To address this flaw, Adam recommends using the following bias corrected values for the first and the second moments:

$$\begin{aligned} \hat{m}_k &= \frac{m_k}{1-{\beta_1}^k} \\ \hat{v}_k &= \frac{v_k}{1-{\beta_2}^k} \end{aligned} \tag{39}$$

- Subsequently, you update the paramter vector p with the following update equation:

$$p_k = p_{k-1} - \gamma \frac{\hat{m}_k}{\sqrt{\hat{v}_k + \epsilon}} \tag{40}$$

- Consult PyTorch's documentation page on the `optim` module for further information regarding the `adam` optimizer.