LAST TIME

- Gradient descent is fragile:
  - Importance of normalization. Discussed batch normalization.
- Overfitting and Regularization
  - Talked about data augmentation and weight-decay

REGULARIZATION

Regularization: Dropout

- Key Idea: Prevent a network from "depending" too much on the presence of a specific activation.
- So, randomly drop these values during training.
  \[ g = \text{Dropout}(f, p) \]

- \( f \) is the incoming activation, \( g \) is the output after dropout. Both will have the same shape.
- \( p \) is a probability (between 0 and 1) that is a parameter of the layer (chosen manually, not learned).

The Dropout "layer" behaves differently during training and testing.

Testing: \( g = f \)

Training

For each element \( f_i \) of \( f \) independently,

- Set \( g_i = 0 \) with probability \( p \).
- Set \( g_i = \alpha f_i \) with probability \( 1 - p \), where \( \alpha \) is a scalar.

What should \( \alpha \) be ?

\[ \alpha = (1 - p)^{-1} \]

so that the \( \mathbb{E} g_i = f_i \), the same as at test time !

- Using dropout forces the network to learn to be robust to deviations from the training set.
  - Forced to learn a fallback even when some activations die.
  - Empirical question of which layers to apply dropout to.
REGULARIZATION

Regularization: Dropout

- Dropout is a layer. You will backpropagate through it! How?
- Write the function as $g = f \cdot \epsilon$
  - Here $\epsilon$ is a random array same size as $f$, with values 0 and $(1 - p)^{-1}$ with probability $p$ and $(1 - p)$.
  - $\cdot$ denotes element-wise multiplication.

So given $\nabla \epsilon$, what is the expression for $\nabla f$?

- $\nabla f = \nabla \epsilon \cdot c$
  - Even though $\epsilon$ is random, you must use the same $\epsilon$ in the backward pass that you generated for the forward pass.
  - Don’t backpropagate $\epsilon$ because it is not a function of the input.
- Like RELU, but kills gradients based on an external random source: whether you dropped that activation or not in the forward pass. If you didn’t, remember to multiply by the $(1 - p)^{-1}$.

REGULARIZATION

Regularization: Early Stopping

- Keep track of dev set error. Stop optimization when it starts going up.
- This is a legitimate regularization technique!
- Essentially, you are restricting your hypothesis space to functions that are reachable within $N$ iterations of a random initialization.

TRAINING IN PRACTICE

Don’t look at this!
**TRAINING IN PRACTICE**

**DIFFERENT OPTIMIZATION METHODS**

- **Standard SGD**
  \[ w_j \leftarrow w_j - \lambda \nabla w_i \]

- **Momentum**
  \[ g_i \leftarrow \nabla w_i + \gamma g_i \]
  \[ w_j \leftarrow w_j - \lambda g_i \]

- But we are still applying the same learning rate for all parameters / weights.

**TRAINING IN PRACTICE**

**DIFFERENT OPTIMIZATION METHODS**

**Adaptive Learning Rate Methods**

Key idea: Set the learning rate for each parameter based on the magnitude of its gradients.

- **Adagrad**
  \[ g_i^2 \leftarrow g_i^2 + (\nabla w_i)^2 \]
  \[ w_j \leftarrow w_j - \lambda \frac{\nabla w_i}{\sqrt{g_i^2 + \epsilon}} \]

Global learning rate divided by sum of magnitudes of past gradients.

Problem: Will always keep dropping the effective learning rate.

- **RMSProp**
  \[ g_i^2 \leftarrow \gamma g_i^2 + (1 - \gamma)(\nabla w_i)^2 \]
  \[ w_j \leftarrow w_j - \lambda \frac{\nabla w_i}{\sqrt{g_i^2 + \epsilon}} \]
DIFFERENT OPTIMIZATION METHODS

Adaptive Learning Rate Methods

- Adam: RMSProp + Momentum

\[
\begin{align*}
m_i &\leftarrow \beta_1 m_i + (1 - \beta_1) \nabla w_i \\
v_i &\leftarrow \beta_2 v_i + (1 - \beta_2) (\nabla w_i)^2 \\
w_i &\leftarrow w_i - \frac{\lambda}{\sqrt{v_i} + \epsilon} m_i
\end{align*}
\]

- How do you initialize \( m_i \) and \( v_i \)? Typically as 0 and 1.
- This won’t matter once the values of \( m_i, v_i \) stabilize. But in initial iterations, they will be biased towards their initial values.

DISTRIBUTED TRAINING

- Neural Network Training is Slow.
- But many operations are parallelizable. In particular, operations for different batches are independent.
- That’s why GPUs are great for deep learning! But even so, you will begin to saturate the computation (or worse, memory) on a GPU.
- Solution: Break up computation across multiple GPUs.
- Two possibilities:
  - Model Parallelism
  - Data Parallelism

DIFFERENT OPTIMIZATION METHODS

Adaptive Learning Rate Methods

- Adam: RMSProp + Momentum + Bias Correction

\[
\begin{align*}
m_i &\leftarrow \beta_1 m_i + (1 - \beta_1) \nabla w_i \\
v_i &\leftarrow \beta_2 v_i + (1 - \beta_2) (\nabla w_i)^2 \\
\hat{m}_i &\leftarrow \frac{m_i}{1 - \beta_1^t} \\
\hat{v}_i &\leftarrow \frac{v_i}{1 - \beta_2^t} \\
w_i &\leftarrow w_i - \frac{\lambda}{\sqrt{\hat{v}_i} + \epsilon} \hat{m}_i
\end{align*}
\]

Here, \( t \) is the iteration number.

As \( t \to \infty \), \( 1 - \beta^2 = 1 \).

DISTRIBUTED TRAINING

Model Parallelism

- Less popular, doesn’t help for many networks.
- Essentially, if you have two independent “paths” in your network, you can place them on different devices. And sync, when they join.

Figure 2: An illustration of the architecture of our CNN, explicitly showing the division of responsibilities between the two GPUs. One GPU owns the layers at the top of the figure while the other owns the layer parts at the bottom. The GPU communicate only at certain layers. The network’s input is 385x385 dimensional, and the number of neurons in the network’s remaining layers is given by 256,144,864,640,128,64,256,4,8,460,6,404,4096,10000.

Was used in the Sutskever et al., 2012 ImageNet paper.
DISTRIBUTED TRAINING

Data Parallelism

- Begin with all devices having the same model weights.
- One each device, load a separate batch of data.
- Do forward-backward to compute weight gradients on each GPU with its own batch.
- Have a single device (one of the GPUs, or a CPU) recover gradients from all devices.
- Average these gradients and apply the update to the weights.
- Distribute new weights to all devices.
- Works well in practice, especially for multiple GPUs in the same machine.
- Communication overhead of transferring gradients and weights back and forth. Can be large if distributing across multiple machines.

- Approximate Distributed Training
  - Let each worker keep updating its own weights independently for multiple iterations. Then, transmit back weights to single device, average weights, and sync to all devices.
  - Other options, quantize gradients when sending back and forth (while making sure all workers have the same models).

NEURAL NETWORKS FOR PHYSICAL TASKS

- So far, we’ve talked about using neural networks for semantic tasks
  - Classification
  - Segmentation
  - Object Detection

- The argument is, semantic tasks have no model to invert, so must rely on learning.
- But remember, for a lot of low and mid-level vision tasks, we also had to use "regularization" or priors.
- These regularizers and priors were hand-crafted. Why not use the expressive capacity of neural networks here?

DENOISING

- Simplest Case
- Traditional Methods
  - Smoothing with a Gaussian Filter
  - Bilateral Filtering
  - Solving some optimization problem

- Instead, just learn a neural network that predicts a clean image from a noisy image.
- Create training samples of noisy and clean pairs \((y_i, x_i)\)
  - Cheap to create. Just get regular images \(x_i\)
  - Synthetically add Gaussian noise with a certain variance to get \(y_i\).
- Train a neural network \(f\) to get \(\hat{x} = f(y)\) and set loss as \(||\hat{x} - x||^2\).
- Learn by back-propagation!

Example: IRCNN. Zhang et al., CVPR 2018.

- Learns a network to predict the noise instead of the clean image
- \(\hat{x} = y + f(y, \theta)\)
DENOISING++

Similarly, networks have been proposed for other image restoration tasks
- Deblurring
- Super-resolution
- In-painting (fill in values in masked out pixels)

But IRCNN proposes a clever way of training networks only for denoising (at different noise levels), and using them for a variety of other tasks—without re-training.

DENOISING++

Plug-and-Play Priors

Problem setup
- You observe \( y = Ax + \) noise.
- You know \( A \). Given \( y \), estimate \( x \).
- \( A \) may not be invertible: the problem is ill-posed

Optimization Setup

\[
\hat{x} = \arg\min_x \| y - Ax \|^2 - \log p(x)
\]

Here \( p(x) \) is a prior on natural clean images \( x \). But we want this to be a complex prior: say something a CNN learns "implicitly".

- You can re-write this by "splitting" the variable \( x \) into \( x \) and \( z \):
  \[
  \hat{x} = \arg\min_{x,z} \| y - Ax \|^2 + \beta(x - z)^2 - \log p(z)
  \]
  
  - Equivalent when \( \beta \to \infty \)

Algorithm

- Initialize \( x \) and pick an initial value of \( \beta \)
- At each iteration
  - Set \( z \) by "denoising" \( x \) with a network that assumes noise variance = \( 1/(2\beta) \).
  - Set \( x \) by minimizing \( \| y - Ax \|^2 + \beta(x - z)^2 \) as a linear least squares problem.
  - Increase \( \beta \) according to some schedule.

This is Denoising: with noise variance \( \sigma^2 = 1/(2\beta) \) ! Use a CNN for this.