**GENERAL**

- Problem Set 5: Deadline Extended to Dec 4th.
- Recitation on Nov 30th (Friday after Thanksgiving)

---

**BATCH NORMALIZATION**

He et al., "Identity Mappings in Deep Residual Networks". 2016.

---

**REGULARIZATION**

- Given a limited amount of training data, deep architectures will begin to overfit.
- **Important**: Keep track of training and dev-set errors. Training errors will keep going down, but dev will saturate. Make sure you don’t train to a point when dev errors start going up.
- So how do we prevent, or delay, overfitting so that our dev performance increases?

Solution 1: Get more data.
Data Augmentation

- Think of transforms to the images that you have that would still keep them in the distribution of real images.
- Typical Transforms
  - Scaling the image
  - Taking random crops
  - Applying Color-transformations (change brightness, hue, saturation randomly)
  - Horizontal Flips (but not vertical)
  - Rotations up to \( \pm 5 \) degrees.
- Are a good way of getting more training data for ‘free’.
- Teaches your network to be invariant to these transformations ....
- Unless your output isn’t. If your output is a bounding box, segmentation map, or other quantities that would change with these augmentation operations, you need to apply them to the outputs too.

Weight Decay

- Add a squared or absolute value penalty on all weight values (for example, on each element of every convolutional kernel, matmul matrix) except biases. \( \sum_i w_i^2 \) or \( \sum_i |w_i| \)
- So now your effective loss is \( L' = L + \lambda \sum_i w_i^2 \)
- How would you train for this ?
  - Let’s say you use backprop to compute \( \nabla_w L \).
  - What gradient would you apply to your weights ? What is \( \nabla_{w_i} L' \) ?
    \[
    \nabla L' = \nabla L + 2\lambda w_i
    \]
- So in addition to the standard update, you will also be subtracting a scaled version of the weight itself.
- What about for \( L' = L + \lambda \sum_i |w_i| \) ?
    \[
    \nabla L' = \nabla L + \lambda \text{Sign}(w_i)
    \]

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.
  \[
g = \text{Dropout}(f, p) ; f \text{ and } g \text{ will have the same shape.}
\]
- Different behavior during training and testing.
  - Training
    - For each element \( f_i \) of \( f \),
      - Set \( g_i = 0 \) with probability \( p \), and \( \frac{f_i}{(1-p)} \) with probability \( (1-p) \)
    - Testing: \( g_i = f_i \)
  - Why does this make sense ? Because in expectation, our value during training and test will be the same.
  - Dropout is a layer. You will backpropagate through it ! How ?

\[
g = f \cdot \epsilon
\]
- Here \( \epsilon \) is a random array same size as \( f \), with values 0 and \( 1/(1-p) \) with probability \( p \) and \( (1-p) \).
- \( \cdot \) denotes element-wise multiplication.
- \( \nabla f = \nabla g \cdot \epsilon \)
  - 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 to \( \epsilon \) 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 \( 1/(1-p) \).
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**

- Train Loss
- Dev Loss
DIFFERENT OPTIMIZATION METHODS

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

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

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

Adaptive Learning Rate Methods

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

- **Adagrad**
  \[ g_t^2 \leftarrow g_t^2 + (\nabla w_t)^2 \]
  \[ w_t \leftarrow w_t - \frac{\lambda}{\sqrt{g_t^2} + \epsilon} \]

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

  Problem: Will always keep dropping the effective learning rate.

- **RMSProp**
  \[ g_t^2 \leftarrow \gamma g_t^2 + (1 - \gamma)(\nabla w_t)^2 \]
  \[ w_t \leftarrow w_t - \frac{\lambda}{\sqrt{g_t^2} + \epsilon} \]

  \[ m_t \leftarrow \beta_1 m_t + (1 - \beta_1)\nabla w_t \]
  \[ v_t \leftarrow \beta_2 v_t + (1 - \beta_2)(\nabla w_t)^2 \]
  \[ w_t \leftarrow w_t - \frac{\lambda}{\sqrt{v_t} + \epsilon} \]

  How do you initialize \( m_t \) and \( v_t \)? Typically as 0 and 1.

  This won’t matter once the values of \( m_t, v_t \) stabilize. But in initial iterations, they will be biased towards their initial values.

- **Adam: RMSProp + Momentum**
  \[ m_t \leftarrow \beta_1 m_t + (1 - \beta_1)\nabla w_t \]
  \[ v_t \leftarrow \beta_2 v_t + (1 - \beta_2)(\nabla w_t)^2 \]
  \[ w_t \leftarrow w_t - \frac{\lambda}{\sqrt{v_t} + \epsilon} \]

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

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

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**
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.

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).

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