LAST TIME

- Talked about the importance of initialization to keep activations (i.e., outputs of layers) “balanced”.
- But initialization only ensures balance at the start of training.
- As your weights update, they can begin to give you biased weights.
- Another option, add normalization in the network itself!


BATCH-NORMALIZATION

- Batch-Norm Layer

\[ y = BN(x) \]
\[ y = \frac{x - \text{Mean}(x)}{\sqrt{\text{Var}(x)}} + \epsilon \]

Here, mean and variance are interpreted per-channel

So for each channel, you compute mean of the values of that channel activation at all spatial locations across all examples in the training set.

But this would be too hard to do in each iteration. So the BN layer just does this normalization over a batch.

And back-propagates through it.

- Batch-Norm Layer

\[ x = (x)_{bije} \]
\[ \mu_c = \frac{1}{BH \sum_{ij} x_{bije}} \]
\[ \sigma^2_c = \frac{1}{BH \sum_{ij} (x_{bije} - \mu_c)^2} \]
\[ y_{bije} = \frac{x_{bije} - \mu_c}{\sqrt{\sigma^2_c} + \epsilon} \]

- The BN layer has no parameters.

What about during back-propagation?

- When you back-propagate through it to \( x \), you go back-propagate through the computation of mean and variance.
- At test time, replace \( \mu_c \) and \( \sigma^2_c \) as mean and variance over the entire training set.
**BATCH-NORMALIZATION**

- Typically apply BN before a RELU.
- Typical use: RELU(BN(Conv(x,k))+b)
  - Don’t add bias before BN as it’s pointless
  - Learn bias post-BN
  - Can also learn a scale: RELU(a BN(Conv(x,k))+b)
- Leads to significantly faster training.
- But you need to make sure you are normalizing over a diverse enough set of samples.

**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.

**REGULARIZATION**

- **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 upto ~ 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.

**REGULARIZATION**

- **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 \( \nabla_w L' \)?
      \[
      \nabla L' = \nabla L + 2\lambda w_i
      \]
  - 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.

\[ g = \text{Dropout}(f, p) \]

- \( f \) 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.

**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.

\[ g = \text{Dropout}(f, p) \]

**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: Dropout**

- Dropout is a layer. You will backpropagate through it! How?
- Write the function as \( g = f \cdot \epsilon \)
  - \( \epsilon \) 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 \( \nabla f \), what is the expression for \( \nabla g \)?

\[ \nabla f = \nabla g \cdot \epsilon \]

- 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 \( (1 - p)^{-1} \).
Train Loss
Dev Loss

Train Loss
Dev Loss

Train Loss
Dev Loss

Train Loss
Dev Loss

Patience is a virtue!
**TRAINING IN PRACTICE**

![Graph 1](image1)

Learning Rate Drops

---

![Graph 2](image2)

Immediate sharp drop, but then rate of improvement slows.

---

**TRAINING IN PRACTICE**

---

**TRAINING IN PRACTICE**

---

**TRAINING IN PRACTICE**

---
**DIFFERENT OPTIMIZATION METHODS**

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

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

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

**Adaptive Learning Rate Methods**

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

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

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

Problem: Will always keep dropping the effective learning rate.

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

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.

\[
\text{As } t \to \infty, 1 - \beta^t = 1.
\]

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.

Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer parts at the top of the figure, while the other runs the layer parts at the bottom. The GPU’s communicate only at certain layers. The network’s input is 193,529-dimensional, and the number of neurons in the network’s commuting layers is given by 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072.

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).