## 1.2 Improving Deep Neural Nets

### 1.2.1 Bias and Variance

- *High bias* means that you aren't fitting the training set well. Use a bigger network or train for longer.
- *High variance* means that there is a big difference between performance on the training and dev set.

### 1.2.2 Regularization

There are several ways to regularize your network:

1. $L_2$ regularization penalizes higher weights:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}||w||_2^2 \ . \tag{3}$$

2. Dropout: randomly set some of the weight values to zero. To avoid scaling output values at test time, use the *inverted dropout* technique, where during training, you divide the output by the `keep_prob`. The intuition is that you can't rely on any one feature, so you have to spread out the weights. Slows down training by a factor of two.
3. Data augmentation.
4. Early stopping is bad because it mixes optimization and not overfitting.

### 1.2.3 Activation functions

1. Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$ is almost never used, except for the output layer in binary classification.
2. Tanh: $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ works better than sigmoid because the activations are closer to zero and you often normalize your input data to have zero mean.
3. ReLU: $a(z) = \max(0, z)$ solves the vanishing gradient problem for positive values and therefore helps the network learn faster. Also, it's derivative is faster to compute. It suffers from the *dying ReLU* problem, however, where a neuron will output zero if it's inputs are negative.
4. Leaky ReLU where $a(z) = \max(0.01z, z)$ gives dying ReLUs a chance to wake up.
5. ELU: $a(z) = \max(e^z - 1, z)$ is the best. It beats Leaky ReLU because it is smooth around $z = 0$, which speeds up gradient descent because it does not bounce as much left and right of $z = 0$. It is slower to compute but compensates for this by its faster convergence rate.

### 1.2.4 Initialization

The more hidden units in a layer, the smaller the initialized weights should be. You can initialize the weights with a normal distribution with mean 0 and standard deviation of:

- For the xavier initialization, use $\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

### 1.2.5 Optimization algorithms

- Mini-batching
  - Say $m = 5,000,000$, then make mini-batches of 1,000 for $t = 5,000$ minibatches. Then for $t = 1$ to $5,000$, gradient descent like usual.
  - Mini-batch size 1 is stochastic gradient descent, where you lose speedup from vectorization.
  - Best mini-batch size typically between 1 and $m$, i.e., 64, 128, 256, 512.

- *Momentum* calculates an exponentially weighted average of examples, i.e., smooths out the steps of gradient descent. For each iteration,
  1. Compute $dW$ for the mini-batch.
  2. $v_{dW} = \beta\ v_{dW} + (1-\beta)dW$ (think of $v_{dW}$ as velocity and $dW$ as acceleration)
  3. $W = W - \alpha\ v_{dW}$
- *RMSprop* aims to dampen oscillations. For each iteration,
  1. Compute $dW$ for the mini-batch.
  2. $s_{dW} = \beta\ s_{dW} + (1-\beta)dW^2$ ($s_{dW}$ is large when $dW$ oscillates a lot)
  3. $W = W - \alpha \frac{dW}{\sqrt{s_{dW}}}$ (slower updates when $s_{dW}$ is large)
- *Adam* (adaptive moment estimation) optimization combines momentum and *RMSprop*, where typically $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Initialize $v_{dw} = 0$, $s_{dw} = 0$. For each iteration $t$,
  1. Compute $dW$ for the mini-batch.
  2. $v_{dW} = \beta_1\ v_{dW} + (1-\beta_1)dW$
  3. $s_{dW} = \beta_2\ s_{dW} + (1-\beta_2)dW^2$
  4. $v_{dW}^{\texttt{corrected}} = \frac{v_{dW}}{1-\beta_1^t}$
  5. $s_{dW}^{\texttt{corrected}} = \frac{s_{dW}}{1-\beta_2^t}$
  6. $W = W - \alpha \frac{v_{dW}^{\texttt{cor}}}{\sqrt{s_{dW}^{\texttt{cor}}}}$
- *Batch normalization* normalizes the hidden layer outputs $z^{[l]}$. Given some intermediate values $z^{(1)}...z^{(m)}$,
  1. Compute mean $\mu = \frac{1}{m}\sum z$ and variance $\sigma^2 = \frac{1}{m}\sum (z-\mu)^2$
  2. Compute the normalized output $z_{\texttt{norm}}^{(i)} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2}}$
  3. Use $\tilde{z}^{(i)} = \gamma\ z_{\texttt{norm}}^{(i)} + \beta$ instead of $z^{(i)}$, where $\gamma$ and $\beta$ are learnable parameters.

  This sort of gets rid of the bias terms and allows the layers to learn independently.
- Learning rate decay: at each epoch, $\alpha_t = 0.95\ \alpha_{t-1}$ .
- Use the *softmax* function for multi-class classification:

$$\texttt{softmax}(y_i) = \frac{e^{y_i}}{\sum e^{y_i}} \tag{4}$$

#### 1.2.6 Q & A

1. **Why does $L_2$ regularization work?** Think of it as making a simpler network. A higher regularization parameter $\lambda$ makes $w \to 0$. And so if you are using a *tanh* activation function, for example, then the activation is more linear at $w \to 0$, so it is harder to form complex decision boundaries that overfit.
2. **Why is $L_2$ regularization better than $L_1$?** $L_1$ creates sparse matrices, because it penalizes smaller values more than $L_2$ does. $L_2$ is more popular.
3. **Why do you need non-linear activation functions?** Because $a^{[2]} = w^{[2]}w^{[1]}x$ and you can express $w^{[2]}w^{[1]}$ as some $w$ so there's no point in multiple hidden layers. I.e., the decomposition of two linear activation functions is a single linear activation function.
4. **How do you train neural networks with imbalanced data?** The simplest approach is to duplicate the imbalanced data to equal your balanced data. More elegant ways of doing this are enforcing balance in each minibatch through sampling, computing weighted gradients, and modifying the loss function.