DEEPLEARNING.AI-C2

Won

COURSE 2

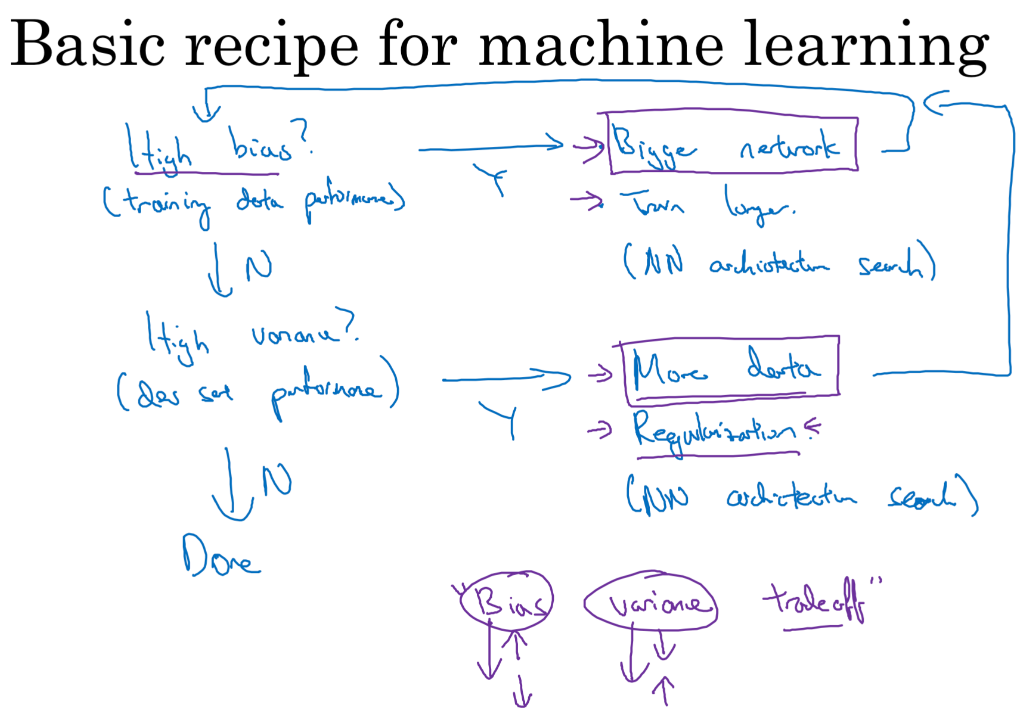
IMPROVING DEEP NEURAL NETWORKS: HYPERPARAMETER TUNING, REGULARIZATION AND OPTIMIZATION

---

train/dev/test sets

Bias / Variance

- high bias, just right, high variance
- Bias "Recipe" for machine learning



Initialization

- *He initialization* -- setting `initialization = "he"` in the input argument. This initializes the weights to random values scaled according to a paper by He et al., 2015.

- If you have heard of "Xavier initialization", this is similar except Xavier initialization uses a scaling factor for the weights $W^{[l]}$ of `sqrt(1./layers_dims[l-1])` where He initialization would use `sqrt(2./layers_dims[l-1])`.
- $\sqrt{\frac{2}{\text{dimension of the previous layer}}}$

What you should remember:

- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't intialize to values that are too large
- He initialization works well for networks with ReLU activations.

Regularization

- **overfitting can be a serious problem**, if the training dataset is not big enough

Regularization Methods

1. **L2 Regularization**
2. **Dropout (Inverted Dropout)**
3. **Data augmentation**
4. **Early Stopping**

L2 Regularization

The standard way to avoid overfitting is called **L2 regularization**. It consists of appropriately modifying your cost function, from:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)$$

To:

$$J_{regularized} = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right) + \frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}$$

backpropagation

- add the regularization term's gradient ($\frac{d}{dW}(\frac{1}{2}\frac{\lambda}{m}W^2) = \frac{\lambda}{m}W$).

Observations:

- The value of λ is a hyperparameter that you can tune using a dev set.
- L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

What is L2-regularization actually doing?:

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

What you should remember - the implications of L2-regularization on:

- The cost computation:
    - A regularization term is added to the cost
- The backpropagation function:
    - There are extra terms in the gradients with respect to weight matrices
- Weights end up smaller ("weight decay"):
    - Weights are pushed to smaller values.

Dropout

- **It randomly shuts down some neurons in each iteration.**
- The idea behind drop-out is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time.
- **Forward propagation with dropout**
- **Backward propagation with dropout**

What you should remember about dropout:

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if keep_prob is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.

Importance

- Regularization will help you reduce overfitting.
- Regularization will drive your weights to lower values.
- **L2 regularization** and **Dropout** are two very effective regularization techniques.

Gradient Checking

- Gradient checking verifies closeness between the gradients from backpropagation and the numerical approximation of the gradient (computed using forward propagation).
- Gradient checking is slow, so we don't run it in every iteration of training. You would usually run it only to make sure your code is correct, then turn it off and use backprop for the actual learning process.

Optimization Methods

- **Exponentially Weighted Averages**
- **RMSprop**
- **Momentum**
- **Adam**
- **Learning Rate Decay**
- **The problem of local optima**

Gradient Descent Rule

The gradient descent rule is, for $l = 1, \ldots, L$:

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]}$$

- where L is the number of layers and α is the learning rate.
- **Gradient Descent**
- **Stochastic Gradient Descent**
- **Mini-Batch Gradient descent**
  - **Shuffling and Partitioning** are the two steps required to build mini-batches
  - Powers of two are often chosen to be the mini-batch size, e.g., 16, 32, 64, 128.

What you should remember:

- The difference between gradient descent, mini-batch gradient descent and stochastic gradient descent is the number of examples you use to perform one update step.
- You have to tune a learning rate hyperparameter α.
- With a well-turned mini-batch size, usually it outperforms either gradient descent or stochastic gradient descent (particularly when the training set is large).

Momentum

- Momentum takes into account the past gradients to smooth out the update.
- The momentum update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases}$$

where L is the number of layers, β is the momentum and α is the learning rate.

- The velocity is initialized with zeros. So the algorithm will take a few iterations to "build up" velocity and start to take bigger steps.

- If $\beta = 0$, then this just becomes standard gradient descent without momentum.

How do you choose β?

- The larger the momentum β is, the smoother the update because the more we take the past gradients into account. But if β is too big, it could also smooth out the updates too much.
- Common values for β range from 0.8 to 0.999. If you don't feel inclined to tune this, $\beta = 0.9$ is often a reasonable default.
- Tuning the optimal β for your model might need trying several values to see what works best in term of reducing the value of the cost function J.

What you should remember:

- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini-batch gradient descent or stochastic gradient descent.
- You have to tune a momentum hyperparameter β and a learning rate α.

Adam

> Adam is one of the most effective optimization algorithms for training neural networks. It combines ideas from **RMSProp** and **Momentum.**

How does Adam work?

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

The update rule is, for $l = 1, \ldots, L$:

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1) \frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v^{corrected}_{dW^{[l]}} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2) (\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s^{corrected}_{dW^{[l]}} = \frac{s_{dW^{[l]}}}{1 - (\beta_1)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v^{corrected}_{dW^{[l]}}}{\sqrt{s^{corrected}_{dW^{[l]}}} + \varepsilon} \end{cases}$$

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.
- α is the learning rate
- ε is a very small number to avoid dividing by zero

Optimization Algorithms

- Mini-batch **Gradient Descent**
- Mini-batch **Adam**

Hyperparameter tuning

- hyperparameters
- try random values
- Coarse to fine
- Normalizing inputs to speed up learning

Softmax regression

Tensorflow

What you should remember:

- Tensorflow is a programming framework used in deep learning
- The two main object classes in tensorflow are **Tensors and Operators**.
- When you code in tensorflow you have to take the following

steps:
  - Create a graph containing Tensors (Variables, Placeholders ...) and Operations (tf.matmul, tf.add, ...)
  - Create a session
  - Initialize the session
  - Run the session to execute the graph
- You can execute the graph multiple times as you've seen in model()
- The backpropagation and optimization is automatically done when running the session on the "optimizer" object.