CRIPAC
智能感知与计算研究中心
Center for Research on Intelligent Perception and Computing

中国科学院自动化研究所
Institute of Automation
Chinese Academy of Sciences

2022

"Deep Learning Lecture"

Lecture 6：Regularization and Optimization

Liang Wang

Center for Research on Intelligent Perception and Computing (CRIPAC)

National Laboratory of Pattern Recognition (NLPR)

Institute of Automation, Chinese Academy of Science (CASIA)

Outline

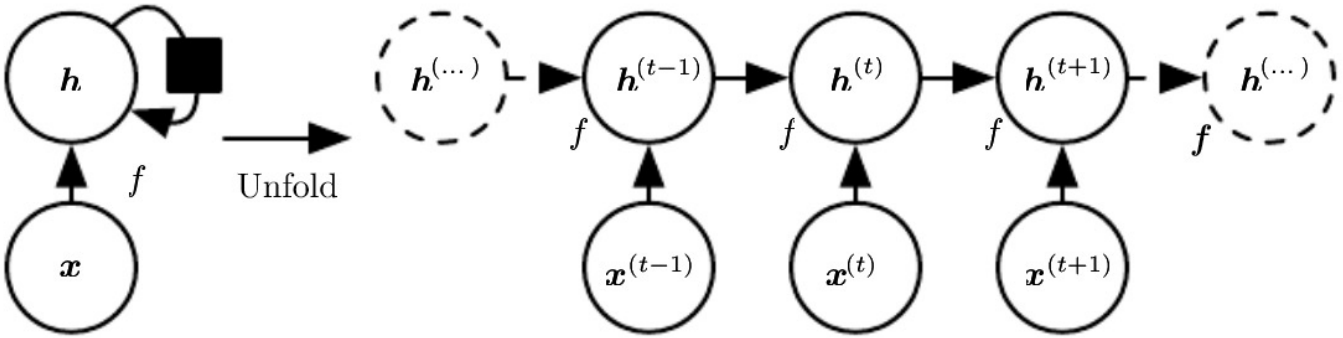
1. Course Review
2. Regularization
3. Optimization
4. Project Grouping

Review: What is an RNN

- We can consider the states to be the hidden units of the network, so we replace $s^{(t)}$ by $h^{(t)}$

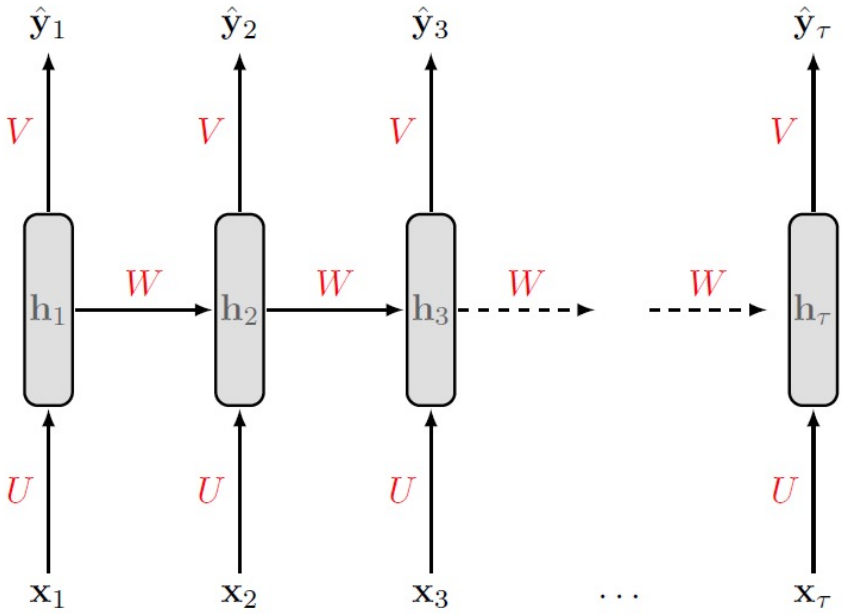
$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

- This system can be drawn in two ways:



- We can have additional architectural features: Such as output layers that read information from h to make predictions.

Review: Basic RNN

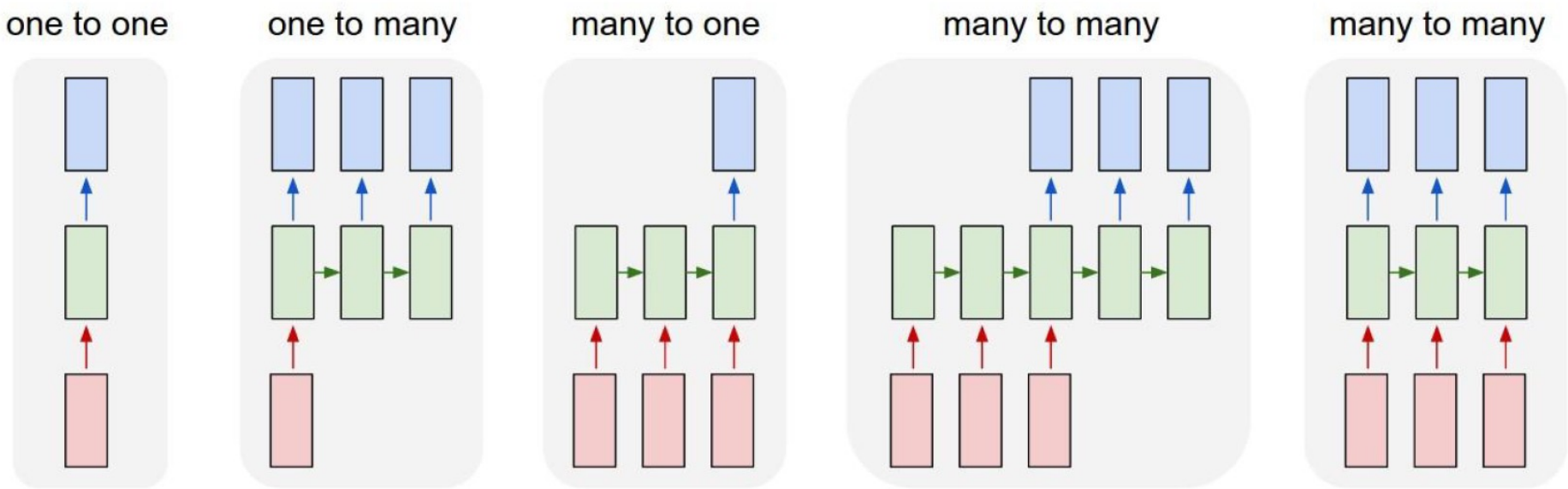


$$\mathbf{a}_t = b + W\mathbf{h}_{t-1} + U\mathbf{x}_t$$
$$\mathbf{h}_t = \tanh \mathbf{a}_t$$
$$\mathbf{o}_t = c + V\mathbf{h}_t$$
$$\hat{\mathbf{y}}_t = \mathsf{softmax}(\mathbf{o}_t)$$

$$L\Big(\{\mathbf{x}_1, \ldots, \mathbf{x}_t\}, \{\mathbf{y}_1, \ldots, \mathbf{y}_t\}\Big) = \sum_t L_t$$

$$\sum_t L_t = -\sum_t \log p_{\mathsf{model}}\big(\mathbf{y}_t | \{\mathbf{x}_1, \ldots, \mathbf{x}_t\}\big)$$

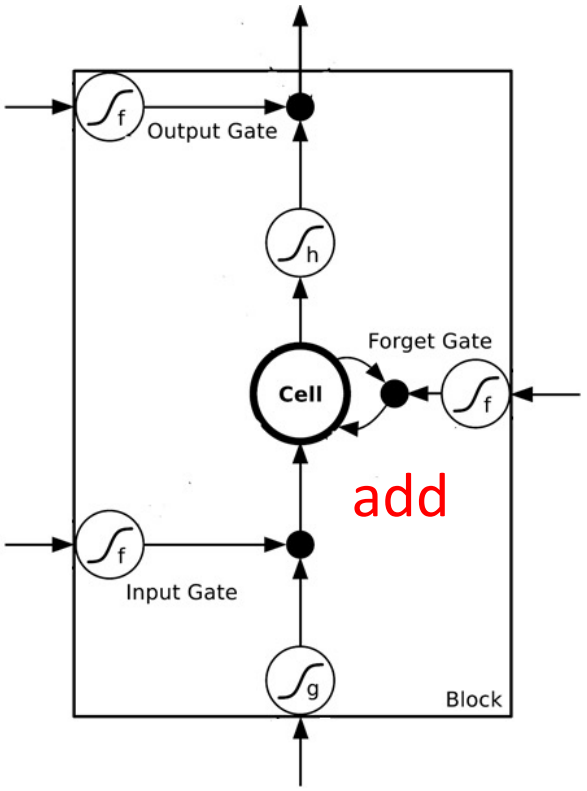
Review: RNN Paradigms



Different problems are more suited for different RNN paradigms

Review: Challenge of Long-Term Dependencies

- Long Short-term Memory (LSTM)
  - Can deal with gradient vanishing (not gradient explode)
    - Memory and input are ***added***
    - The influence never disappears unless forget gate is closed



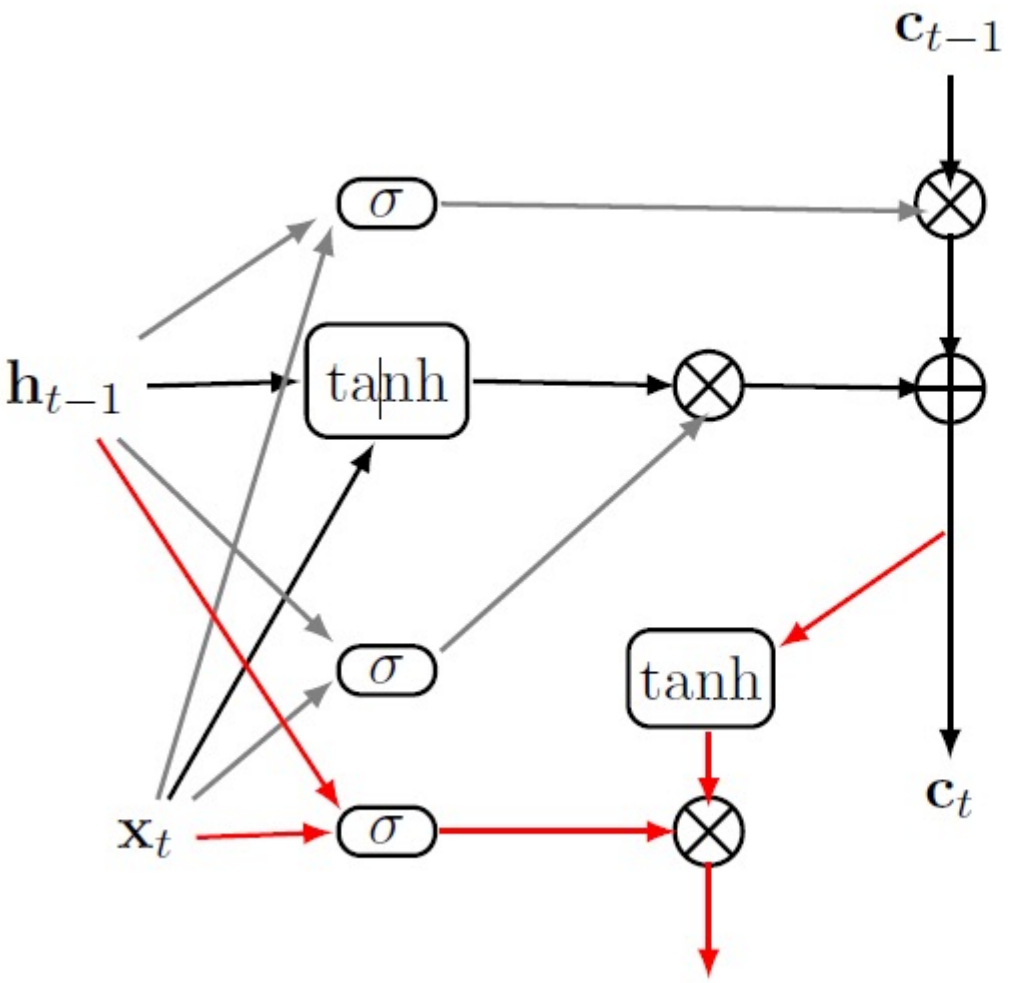
And:

Gated Recurrent Unit (GRU): simpler than LSTM

[Cho, EMNLP'14]

Helpful Techniques!

Review: Long Short Term Memory (LSTM)



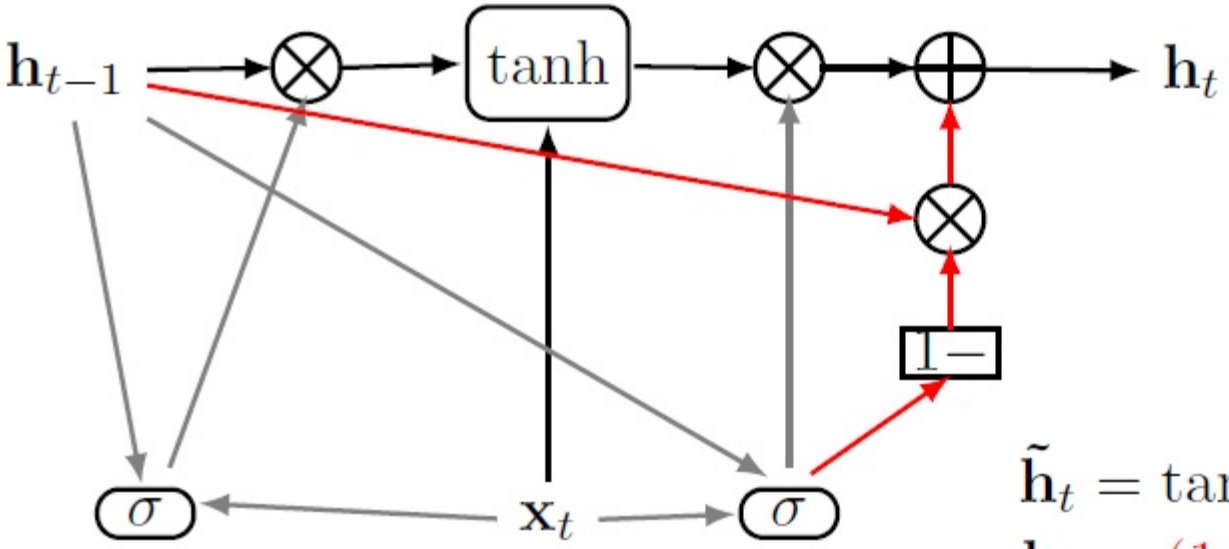
$$f_t = \sigma(W_f \mathbf{h}_{t-1} + U_f \mathbf{x}_t)$$
$$i_t = \sigma(W_i \mathbf{h}_{t-1} + U_i \mathbf{x}_t)$$
$$o_t = \sigma(W_o \mathbf{h}_{t-1} + U_o \mathbf{x}_t)$$

$$\tilde{\mathbf{c}}_t = \tanh(W \mathbf{h}_{t-1} + U \mathbf{x}_t)$$
$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + i_t \odot \tilde{\mathbf{c}}_t$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

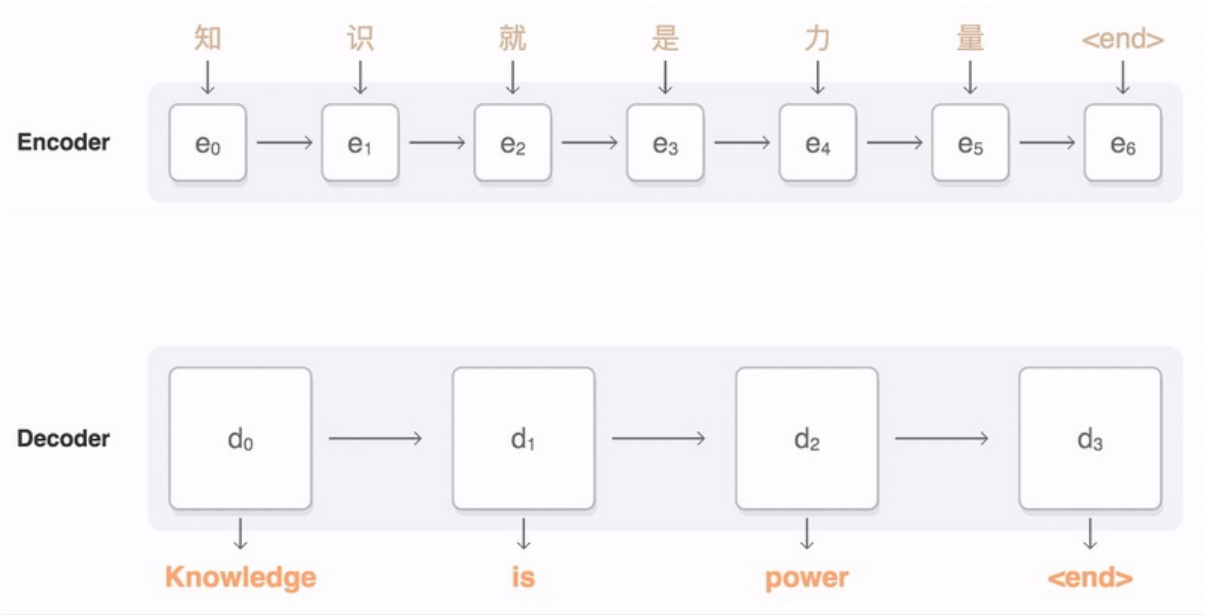
Review: Gated Recurrent Unit (GRU)

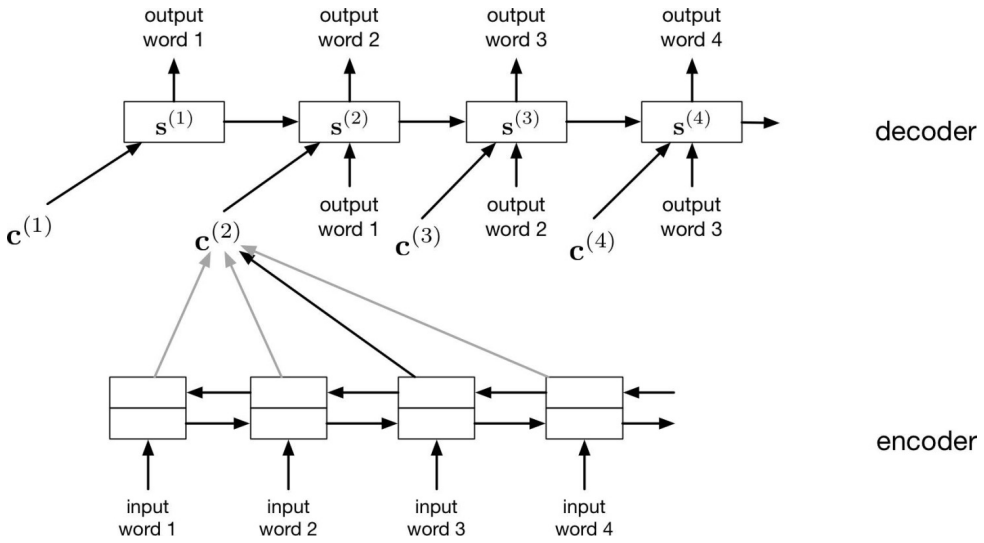
$$r_t = \sigma(W_r \mathbf{h}_{t-1} + U_r \mathbf{x}_t)$$
$$z_t = \sigma(W_z \mathbf{h}_{t-1} + U_z \mathbf{x}_t)$$



$$\tilde{\mathbf{h}}_t = \tanh(W(r_t \odot \mathbf{h}_{t-1}) + U\mathbf{x}_t)$$
$$\mathbf{h}_t = (1 - z_t) \odot \mathbf{h}_{t-1} + z_t \odot \tilde{\mathbf{h}}_t$$

Review: Neural Machine Translation





Review: Image Captioning

test image

image

conv-64
conv-64
maxpool

conv-128
conv-128
maxpool

conv-256
conv-256
maxpool

conv-512
conv-512
maxpool

conv-512
conv-512
maxpool

FC-4096
FC-4096

v

y0 y1 y2

h0 h1 h2

x0 <START>

straw

hat

<START>

sample
<END> token
=> finish.

$$h_t = \tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + W_{ih} \cdot v)$$

Outline

1. Course Review
2. Regularization
3. Optimization
4. Project Grouping

Review Some Important Concepts

Generalization

- Generalization error: the expected error over **ALL** examples
- To obtain theoretical guarantees about generalization of a machine learning algorithm, we assume all the samples are drawn from a distribution p(x,y), and calculate generalization error (GE) of a prediction function f by taking expectation over p(x, y)

$$GE_f = \int_{\mathbf{x},y} p(\mathbf{x}, y)\mathbf{Error}(f(\mathbf{x}), y)$$

- However, in practice, p(x,y) is unknow. We assess the generalization performance with a test set

$$\text{Performance}_{\text{test}} = \frac{1}{M}\sum_{i=1}^{M} \text{Error}(f(\mathbf{x}_i^{(\text{test})}), y_i^{(\text{test})})$$

- We hope that both test examples and training examples are drawn from p(x, y) of interest, although it is unknown.

Underfitting Vs Overfitting

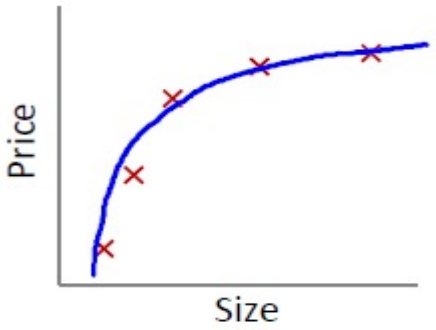
Underfitting

- The learner cannot find a solution that fits training examples well. Underfitting means the learner cannot capture some important aspects of the data.
  - For example, use linear regression to fit training examples $\left\{x_i^{(train)}, y_i^{(train)}\right\}$, where $y_i^{(train)}$ is an quadratic function of $x_i^{(train)}$.

Overfitting

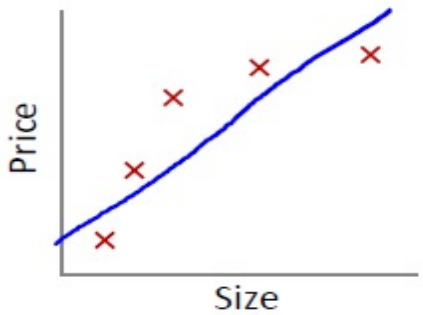
- The learner fits the training data well, but loses the ability to generalize well, i.e. it has small training error but larger generalization error. A learner with large capacity tends to overfit.

Underfitting Vs Overfitting



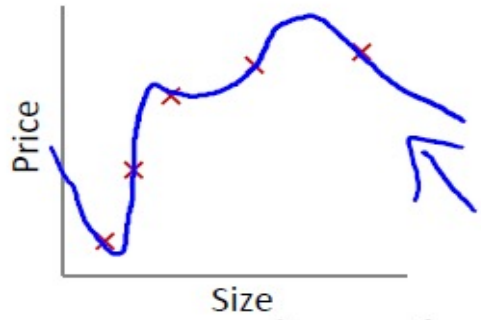
$\rightarrow \theta_0 + \theta_1 x + \theta_2 x^2$

"Just right"



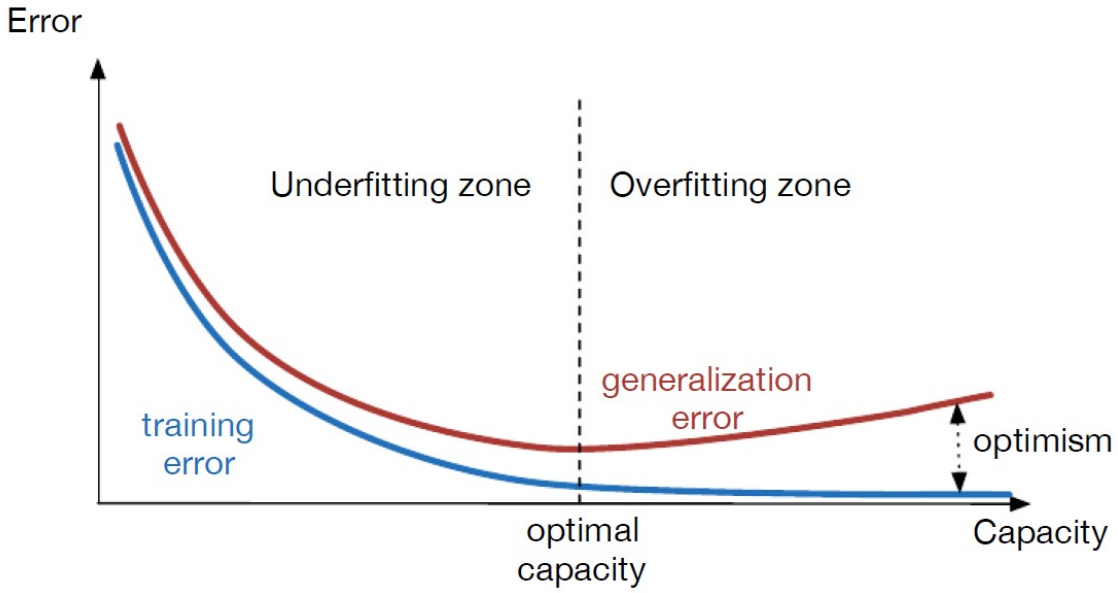
$\rightarrow \theta_0 + \theta_1 x$

"Underfit" "High bias"



$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

"Overfit" "High variance"



Bias

$$\text{bias}(\hat{\theta}) = E(\hat{\theta}) - \theta$$

where expectation is over all the train sets of size n sampled from the underlying distribution

- An estimator is called unbiased if $E(\hat{\theta}) = \theta$
- Example: Gaussian distribution. $p(\mathbf{x}_i; \theta) = \mathcal{N}(\theta, \Sigma)$ and the estimator is $\hat{\theta} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i^{(\text{train})}$

$$E(\hat{\theta}) = E\left[\frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i^{(\text{train})}\right] = \frac{1}{n} \sum_{i=1}^{n} E\left[\mathbf{x}_i^{(\text{train})}\right] = \frac{1}{n} \sum_{i=1}^{n} \theta = \theta$$

Variance

$$\text{Var}[\hat{\theta}] = E[(\hat{\theta} - E[\hat{\theta}])^2] = E[\hat{\theta}^2] - E[\hat{\theta}]^2$$

- Variance typically decreases as the size of the train set increases.
- Both bias and variance are the sources of estimation errors.

$$\text{MSE} = E[(\hat{\theta} - \theta)^2] = \text{Bias}(\hat{\theta})^2 + \text{Var}[\hat{\theta}]$$

- Increasing the capacity of a learner may also increase variance, although it has better chance to cover the true function.

Regularization

Introduction

- **Regularization** is any modification made to the learning algorithm with an intention to lower the generalization error but not the training error.
- Many standard regularization concepts from machine learning can be readily extended to deep models
- In context of deep learning, most regularization strategies are based on **regularizing estimators**. This is done through reducing variance at the expense of increasing the bias of the estimator.
- An effective regularizer is one that decreases the variance significantly while not overly increasing the bias.

Introduction

- We discussed three regimes concerning the capacity of models where the model either:
  - —Excludes the true data generating process which induces bias (under fitting).
  - —Matches the true data generating process.
  - —Includes the true data generating process, but also includes many other possible candidates, which results in variance dominating the estimation error (over fitting).
- The goal of regularization is to take the model from the **third** to the **second** regime**.**

Motivation

- In practice, we never have access to the true data generating distribution. This is a direct result of the extremely complicated domains (images, text and audio sequences) we work with when applying deep learning algorithms.
- In most applications of deep learning, the data generating process is almost certainly **outside the chosen model family.**
- All of the above implies that controlling the complexity of the model is not a simple matter of finding the right model size and the right number of parameters.
- Instead, deep learning relies on finding the best fitting model as a large model that has been **regularized** properly.

Classical Regularization Strategies

Parameter Norm Penalties

- The most traditional form of regularization applicable to deep learning is the concept of **parameter norm penalties.**
- This approach limits the capacity of the model by adding the penalty $\Omega(\theta)$ to the objective function resulting in:

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(\theta)$$

- $\alpha \in [0,1)$ is a hyper-parameter that weights the relative contribution of the norm penalty to the value of the objective function.

Parameter Norm Penalties

- When the optimization procedure tries to minimize the objective function, it will also decrease some measure of **size** of the parameters.
- Note: The bias terms in the affine transformations of deep models usually require less data to be fit and are usually left unregularized.
- Without loss of generality, we will assume we will be regularizing only the weights **w.**

L2 Norm Parameter Regularization

- The L2 parameter norm penalty, also known as **weight decay** drives **w** closer to the origin by adding the regularization term:

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(\theta)$$

$$\Omega(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||_2^2 = \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

- For now, assume there is no bias parameters, only weights.

L2 Regularization: Effect on gradient descent

- Gradient of regularized objective

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

- Gradient descent update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})).$$

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$$

- The weights **multiplicatively shrink** by a constant factor at each step.

L2 Regularization: Effect on the optimal solution

- Consider a quadratic approximation around w^*

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*),$$

- The minimum of $\hat{J}$ occurs where its gradient is equal to 0

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*)$$

- Gradient of regularized objective

$$\alpha\tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) = 0$$

$$(\boldsymbol{H} + \alpha\boldsymbol{I})\tilde{\boldsymbol{w}} = \boldsymbol{H}\boldsymbol{w}^*$$

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha\boldsymbol{I})^{-1}\boldsymbol{H}\boldsymbol{w}^*$$

- $\widetilde{w}$ represent the location of the minimum

L2 Regularization: Effect on the optimal solution

- $\boldsymbol{H}$ is real and symmetric

$$\begin{aligned}\tilde{\boldsymbol{w}} &= (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top + \alpha\boldsymbol{I})^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^* \\ &= \left[\boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})\boldsymbol{Q}^\top\right]^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^* \\ &= \boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^*.\end{aligned}$$

- Effect: rescale along eigenvectors of $\boldsymbol{H}$

L1 Norm Parameter Regularization

- L1 norm is another option that can be used to penalize the size of model parameters.
- L1 regularization on the model parameters **w** is:

$$\Omega(\theta) = ||\mathbf{w}|| = \sum_i |w_i|$$

- the regularized objective function

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \|\boldsymbol{w}\|_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

L1 Regularization: Effect on gradient descent

- Gradient of regularized objective

$$\nabla_w \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \text{sign}(\boldsymbol{w}) + \nabla_w J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

- What is the difference between L2 and L1 norm penalty ?
  - the regularization contribution to the gradient no longer scales linearly with each w_i, it is a constant factor with a sign equal to $sign(w_i)$

L1 Regularization: Effect on the optimal solution

- This is a truncated Taylor series approximating the cost function of a more sophisticated model. The gradient in this setting is given by:

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*),$$

- Further assume that $\boldsymbol{H}$ is diagonal and positive. The regularized objective is

$$\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}^*; \boldsymbol{X}, \boldsymbol{y}) + \sum_i \left[\frac{1}{2} H_{i,i} (w_i - w_i^*)^2 + \alpha |w_i| \right].$$

L1 Regularization: Effect on the optimal solution

- The problem of minimizing this approximate cost function has an analytical solution (for each dimension i), with the following form:

$$w_i = \text{sign}(w_i^*) \max\left\{|w_i^*| - \frac{\alpha}{H_{i,i}}, 0\right\}.$$

- Consider the situation where $w_i^* > 0$ for all i. There are two possible outcomes:
  - $w_i^* \leq \frac{\alpha}{H_{ii}}$. Here the optimal value of w_i under the regularized objective is simply $w_i = 0$
  - $w_i^* > \frac{\alpha}{H_{ii}}$. In this case, the regularization does not move the optimal value of w_i to zero but instead just shifts it in that direction by a distance equal to $\frac{\alpha}{H_{ii}}$

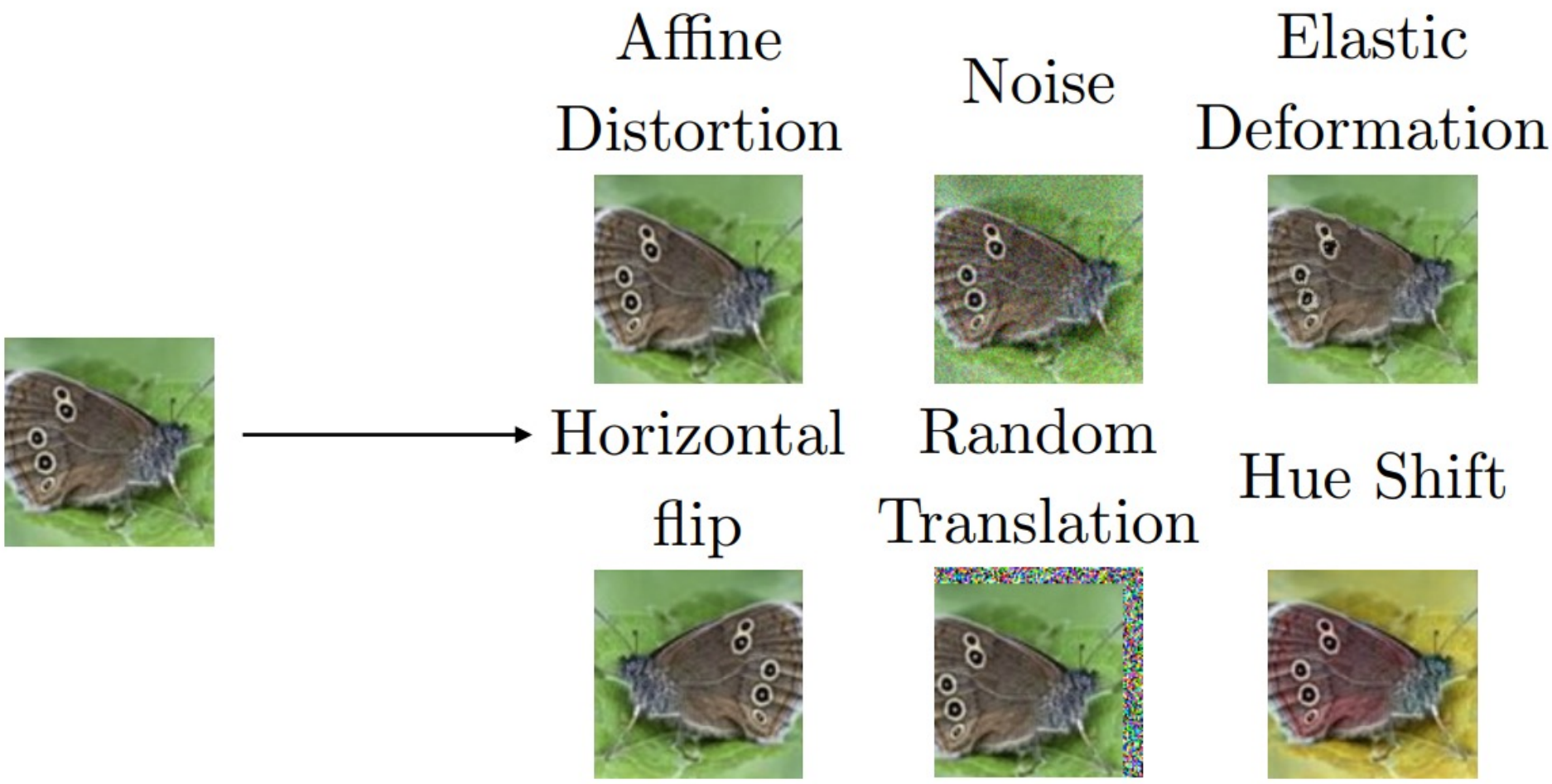
L1 Norm Parameter Regularization

- In comparison to L2 regularization, L1 regularization results in a solution that is more **sparse**.
- This sparsity property can be thought of as a feature selection mechanism.

Dataset Augmentation

- We have seen that for consistent estimators, the best way to get better generalization is to train on more data.
- The problem is that under most circumstances, data is limited. Furthermore, labelling is an extremely tedious task.
- **Dataset Augmentation** provides a cheap and easy way to increase the amount of your training data.
- Certain tasks such as steering angle regression require dataset augmentation to perform well.

Dataset Augmentation



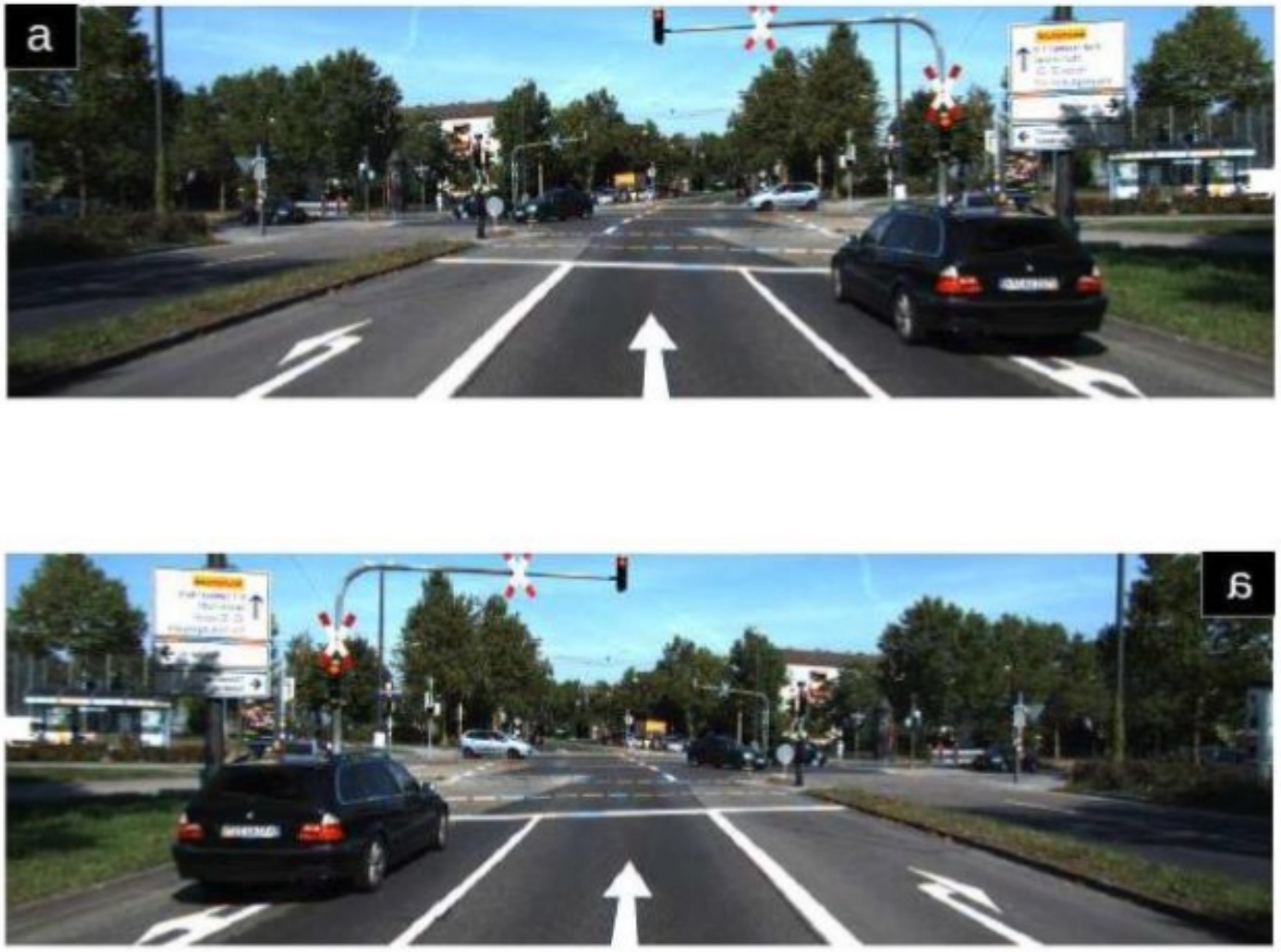
Dataset Augmentation: Color jitter

Dataset Augmentation: Color jitter

- **Color jitter** is a very effective method to augment datasets. It is also extremely easy to apply.
- **Fancy PCA** was proposed by Krizhevsky *et al*. in the famous Alex net paper. It is a way to perform color jitter on images.
- Fancy PCA Algorithm:
  - Perform PCA on the three color channels of your entire dataset.
  - From the covariance matrix provided by PCA, extract the eigenvalues λ_1, λ_2, λ_3 and their corresponding eigenvectors p_1, p_2, p_3.
  - Add $p_i\,[a_1\,\lambda_1, a_2\,\lambda_2, a_3\,\lambda_3]^T$ to the i^{th} color channel. $a_1 \ldots a_3$ *are* random variables sampled for each augmented image from a zero mean Gaussian distribution with a variance of 0.1.

A. Krizhevsky, I. Sutskever, and G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012.

Dataset Augmentation: Horizontal Flipping



Dataset Augmentation: Horizontal Flipping

- **Horizontal Flipping** is applied on data that exhibit horizontal asymmetry.
- Horizontal flipping can be applied to natural images and point clouds. Essentially, one can double the amount of data through horizontal flipping.

Dataset Augmentation: Conclusion

- Many other task specific dataset augmentation algorithms exist. It is highly advised to always use dataset augmentation.
- However, be careful not to alter the correct output!
  - Example: b and d, horizontal flipping.
- Furthermore, when comparing two machine learning algorithms, train them both with either augmented or non-augmented dataset. Otherwise, no subjective decision can be made on which algorithm performed better.

Noise Robustness

- **Noise Injection** can be thought of as a form of regularization. The addition of noise with infinitesimal variance at the input of the model is equivalent to imposing a penalty on the norm of the weights (Bishop, 1995).
- Noise can be injected at different levels of deep models.

Noise Robustness: Noise Injection on Weights

- Noise added to weights can be interpreted as a more traditional form of regularization.
- This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output.
- In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions (Hochreiter and Schmidhuber, 1995).

Noise Robustness: Noise Injection on Outputs

- Most datasets have some (A LOT!) mistakes in the y labels. Minimizing our cost function on wrong labels can be extremely harmful.
- One way to remedy this is to explicitly model the noise on labels. This is done through setting a probability ε for which we think the labels are correct.
- This probability is easily incorporated into the cross entropy cost function **analytically.**
- An example is **label smoothing.**

Noise Robustness : Label Smoothing

- Usually, we have output vectors provided to us as
  - y_{label} = [1, 0, 0, 0...0].
- Softmax output is usually of the form
  - y_{out} = [0.87, 0.001, 0.01, 0.1, ...0.01].
- Maximum likelihood learning with a softmax classifier and hard targets may actually never converge, the softmax can never predict a probability of exactly 0 or exactly 1, so it will continue to learn larger and larger weights, making more extreme predictions.

Noise Robustness : Label Smoothing

- **Label smoothing** replaces the label vector with

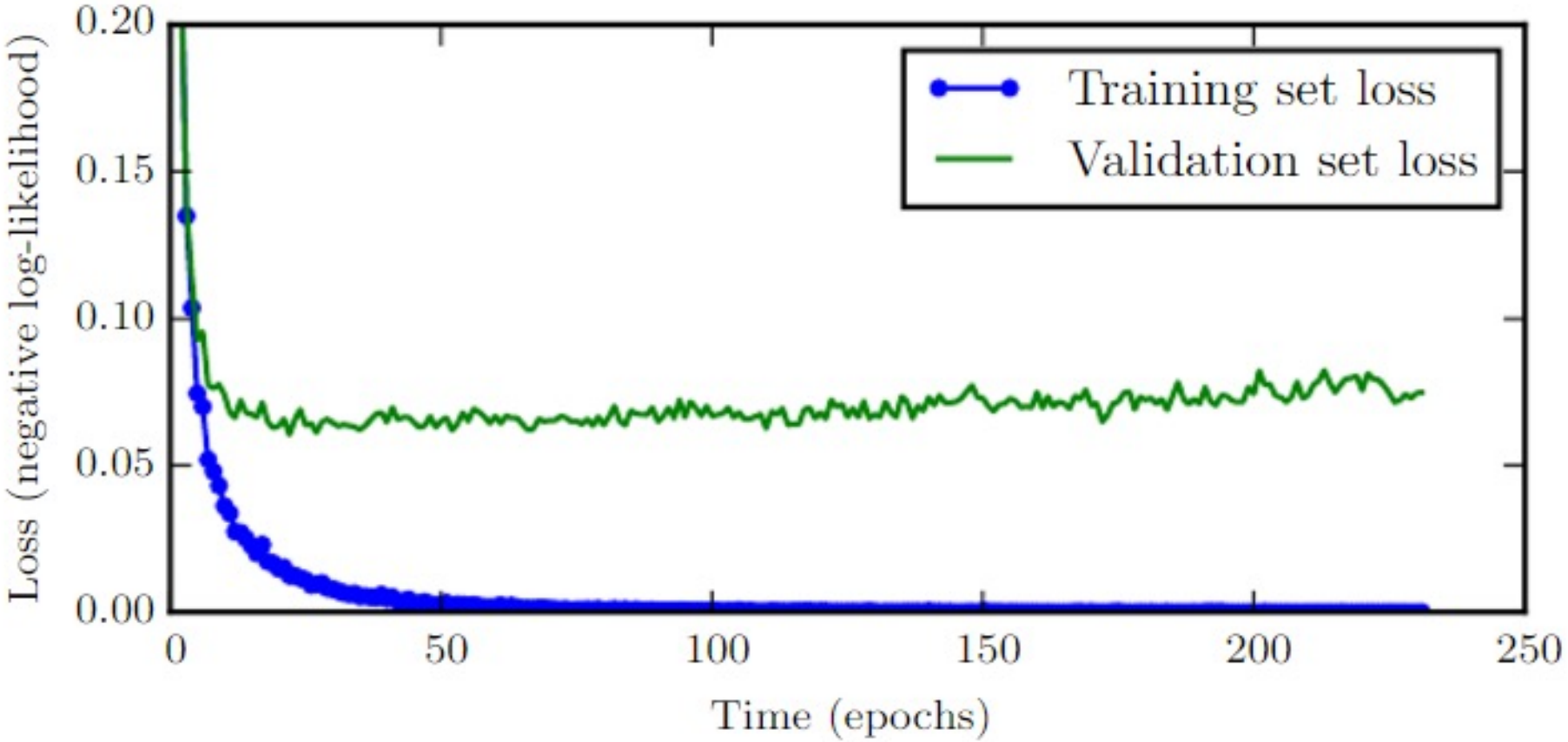
$$y_{label} = [1 - \epsilon, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1} \cdots \frac{\epsilon}{K-1}]$$

- The above representation has the advantage of preventing the pursuit of hard probabilities without discouraging correct classification.

Early Stopping: Motivation

- When training models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, while the error on the validation set begins to rise again.
- The occurrence of this behaviour in the scope of our applications is almost certain.
- This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error.
- This is termed **Early Stopping.**

Early Stopping



Early Stopping: Practical Issues

- Early Stopping is probably one of the most used regularization strategies in deep learning.
- Early stopping can be thought of as a hyper-parameter selection method, where **training time** is the hyper-parameter to be chosen.
- However, a portion of data should be reserved for validation.

Early Stopping: Exploiting the Validation Data

- To exploit all of our precious training data we can:
  - Employ early stopping as described above.
  - Retrain using all of the data up to the point that was determined during early stopping.
- Some subtleties arise regarding the definition of **point.**
- Do we train for the same number of parameter updates or for the same number of epochs (passes through training data) ?

Early Stopping: Exploiting the Validation Data

- A second strategy to exploit the full training dataset would be to:
  - Employ early stopping as described above.
  - Continue training with the parameters determined by early stopping, using the validation set data.
- This strategy avoids the high cost of retraining the model from scratch, but is not well-behaved.
- Since we no longer have a validation set, we cannot know if generalization error is improving or not. Our best bet is to stop training when the training error is not decreasing much any more.

Parameter Sharing

- So far, we have discussed regularization as adding constraints or penalties to the parameters with respect to a **fixed region.**
- However, we might want to express priors on parameters in other ways. Specifically, we might not know which region the parameters would lie in, but rather that there is some dependencies between them.
- Most common type of dependency: Some parameters should be close to each other.

Parameter Sharing

- **Parameter Sharing** imposes much stronger assumptions on parameters through forcing the parameter sets to be **equal.**
- Examples would be Siamese networks, convolution operators, and multitask learning.

Parameter Tying

- **Parameter Tying** refers to explicitly forcing the parameters of two models to be close to each other, through the norm penalty:

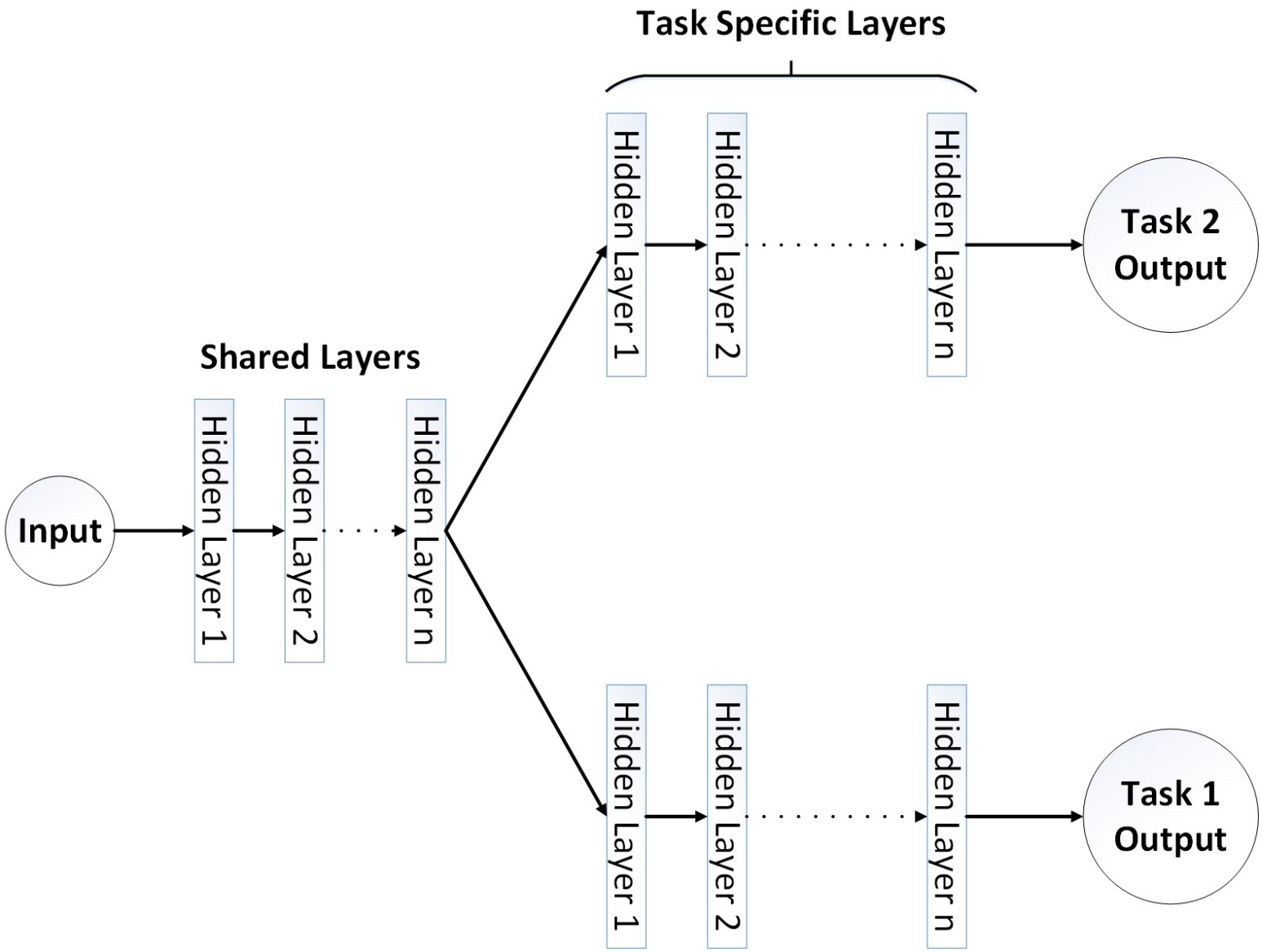
$$||\mathbf{w}^{(A)} - \mathbf{w}^{(B)}||$$

- Here, $w^{(A)}$ refers to the weights of the first model while $w^{(B)}$ refers to those of the second one.

Multitask Learning

- **Multitask Learning** is a way to improve generalization by pooling the examples arising out of several tasks.
- Usually, the most common form of multitask learning is performed through an architecture which is divided to two parts:
  - Task-specific parameters (which only benefit from the examples of their task to achieve good generalization).
  - Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks).
- Multitask learning is a form of parameter sharing.

Multitask Learning



Multitask Learning

- Improved generalization and generalization error bounds (Baxter, 1995) can be achieved because of the shared parameters, for which statistical strength can be greatly improved in proportion with the increased number of examples for the shared parameters, compared to the scenario of single-task models.
- Intuitively, the additional task imposes constraints on the parameters in the shared layers, preventing overfitting.
- Improvement in generalization only occurs when there is something shared across the tasks at hand.

Bagging

- **Bagging** (short for **bootstrap aggregating**) is a technique for reducing generalization error through combining several models (Breiman, 1994).
- Bagging is defined as follows:
  - Train *k* different models on *k* different subsets of training data, constructed to have the same number of examples as the original dataset through random sampling from that dataset with replacement.
  - Have all of the models vote on the output for test examples.
- Techniques employing bagging are called ensemble models.

Bagging

- The reason that Bagging works is that different models will usually not all make the same errors on the test set.
- This is a direct results of training on *k* different subsets of the training data, where each subset is missing some of the examples from the original dataset.
- Other factors such as differences in random initialization, random selection of mini-batches, differences in hyperparameters, or different outcomes of non-deterministic implementations of neural networks are often enough to cause different members of the ensemble to make **partially independent errors.**

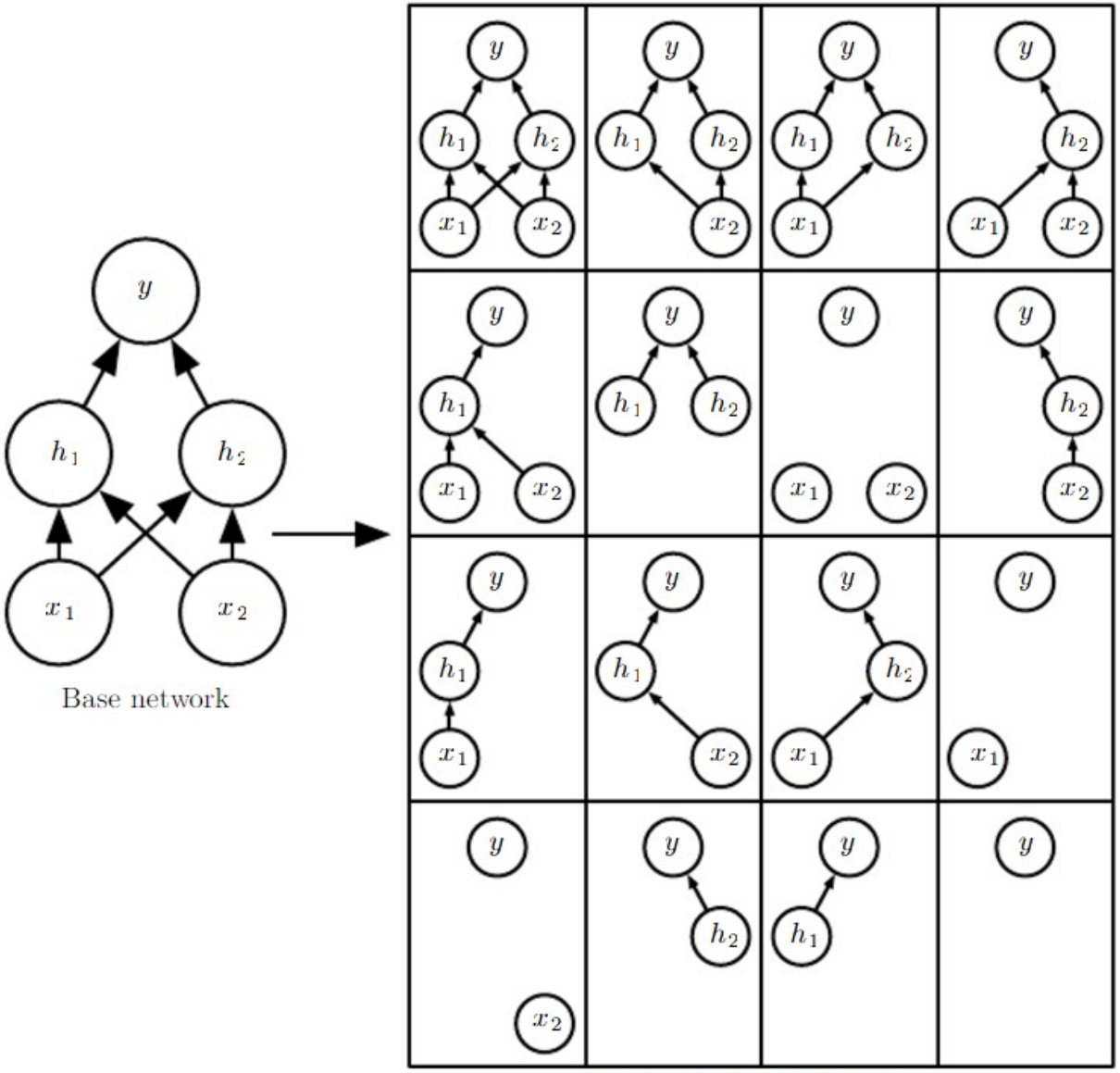
Ensemble Models

- On average, the ensemble will perform at least as well as any of its members, and if the members make independent errors, the ensemble will perform significantly better than its members.
- The only disadvantage of ensemble models is that they do not provide us with a **scalable** way to improve performance**.** Usually, ensemble models of more than 2-3 networks become too tedious to train and handle.

Dropout

- Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models.
- Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.
- Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network.

Dropout



Dropout

- Dropout allows us to represent an exponential number of models with a tractable amount of memory.
- Furthermore, Dropout removes the need to accumulate model votes at the inference stage.
- Dropout can intuitively be explained as forcing the model to learn with missing input and hidden units.

Training with Dropout

- To train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent.
- Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network.
- The mask for each unit is sampled independently from all of the others.
- Typically, the probability of including a hidden unit is 0.5, while the probability of including an input unit is 0.8.

Training with Dropout

- Dropout training has some intricacies we need to be wary of.
- At training time, we are **required** to divide the output of each unit by the probability of that unit's dropout mask.
- The goal is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average.
- No theoretically satisfying basis for the accuracy of this approximate training rule in deep non linear networks, but empirically it performs very well.

Conclusion

- Dropout is that it is very computationally cheap, using dropout during training requires only *O(n)* computation per example per update, to generate *n* random binary numbers and multiply them by the state.
- Dropout does not significantly limit the type of model or training procedure that can be used. It works well with nearly any model that uses a distributed representation and can be trained with stochastic gradient descent.

Conclusion

- Though the cost per-step of applying dropout to a specific model is negligible, the cost of using dropout in a complete system can be significant.
- Applying Dropout indirectly requires us to design larger systems to preserve capacity. Larger systems usually are slower at inference time.
- Practitioners have to keep in mind that for very large datasets, regularization confers little reduction in generalization error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularization.

Adversarial Training

- In many cases, neural networks have begun to reach human performance when evaluated on an i.i.d. test set.
- However, szegedy *et al*.(2014) found that even networks that have achieved human accuracy, have a 100% error rate on examples that have been intentionally constructed to ”fool” the network.
- In many cases, the modified example is so similar to the original one, human observers cannot tell the difference.
- These examples are called **adversarial examples.**

Adversarial Examples

$\boldsymbol{x}$

$y =$"panda"
w/ 57.7%
confidence

$+ .007 \times$

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"nematode"
w/ 8.2%
confidence

$=$

$\boldsymbol{x} +$
$\epsilon \, \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"gibbon"
w/ 99.3 %
confidence

Adversarial Examples

- Adversarial examples are interesting in the context of regularization because one can reduce the error rate on the original i.i.d. test set via **adversarial training** - training on adversarially perturbed examples from the training set.
- Adversarial training discourages highly sensitive linear behaviour through explicitly introducing a local constancy prior into supervised neural nets.

Outline

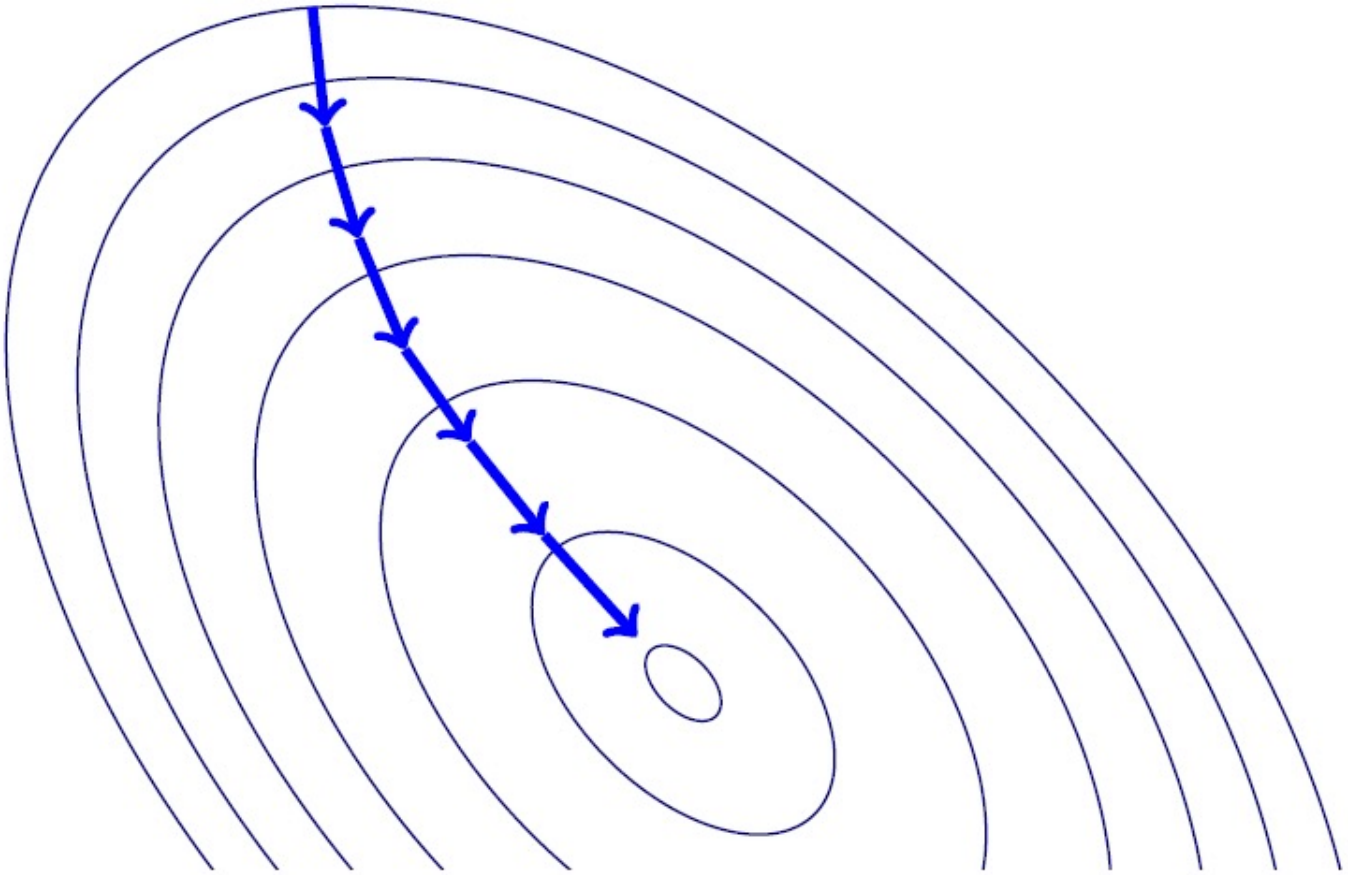
1. Course Review
2. Regularization
3. Optimization
4. Project Grouping

Optimization

- We've seen backpropagation as a method for computing gradients

- Let's see a family of first order methods

Gradient Descent



Batch Gradient Descent

Algorithm 1 Batch Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**

2: Compute gradient estimate over N examples:

3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$

4: Apply Update:

5: **end while** $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update

Stochastic Gradient Descent

Algorithm 2 Stochastic Gradient Descent at Iteration k

Require: Learning rate ϵ_k

Require: Initial Parameter θ

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

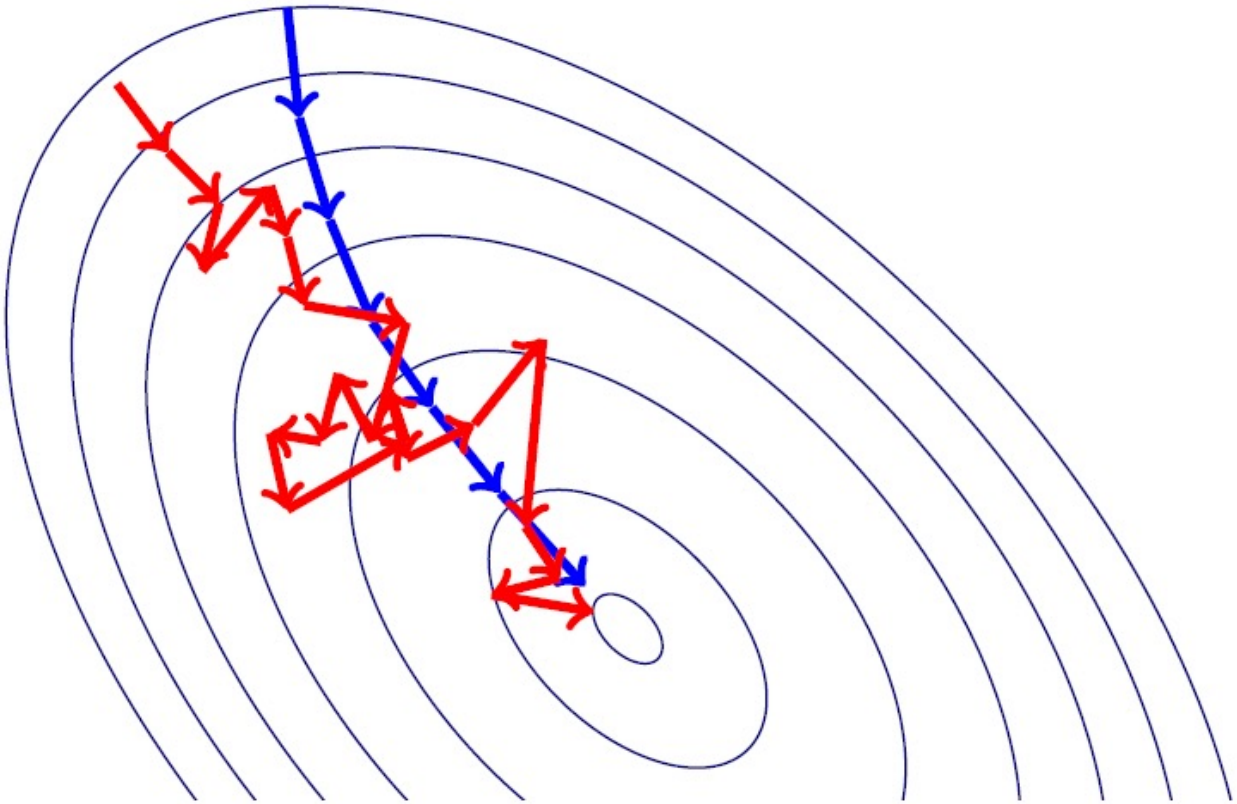
3: Compute gradient estimate:

4: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

5: **end while** $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$

- ϵ_k is learning rate at step k

Stochastic Gradient Descent



Learning Rate Schedule

- In practice the learning rate is decayed linearly till iteration τ

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \quad \text{with} \quad \alpha = \frac{k}{\tau}$$

- τ is usually set to the number of iterations needed for a large number of passes through the data
- ϵ_τ should roughly be set to 1% of ϵ_0
- How to set ϵ_0 ?

Minibatching

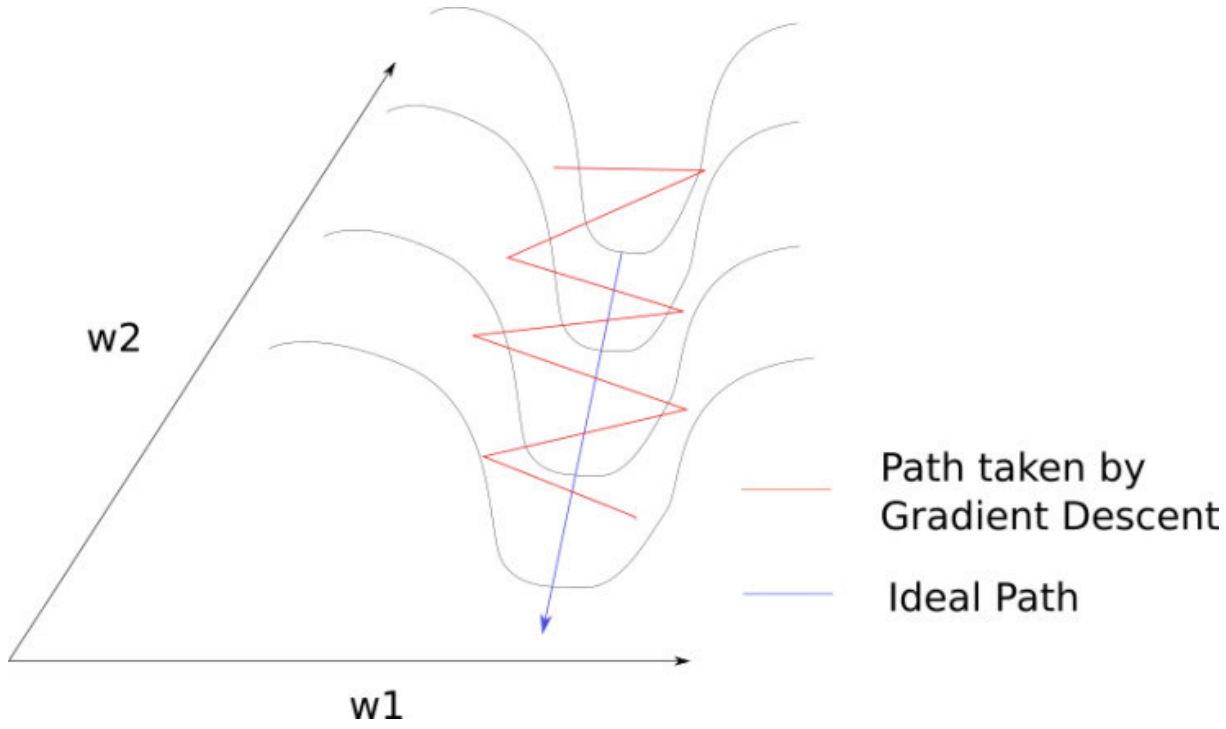
- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches
- Advantage: Computation time per update does not depend on number of training examples N
- This allows convergence on extremely large datasets
- See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou

Momentum

- The Momentum method is a method to accelerate learning using SGD

- In particular SGD suffers in the following scenarios:
  - Error surface has high curvature
  - We get small but consistent gradients
  - The gradients are very noisy

Momentum



- Gradient Descent would move quickly down the walls, but very slowly through the valley floor

Momentum

- How do we try and solve this problem?
- Introduce a new variable v, the velocity
- We think of v as the direction and speed by which the parameters move as the learning dynamics progresses
- The velocity is an exponentially decaying moving average of the negative gradients

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left(L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$$

- $\alpha \in [0, 1)$ Update rule: $\theta \leftarrow \theta + \mathbf{v}$

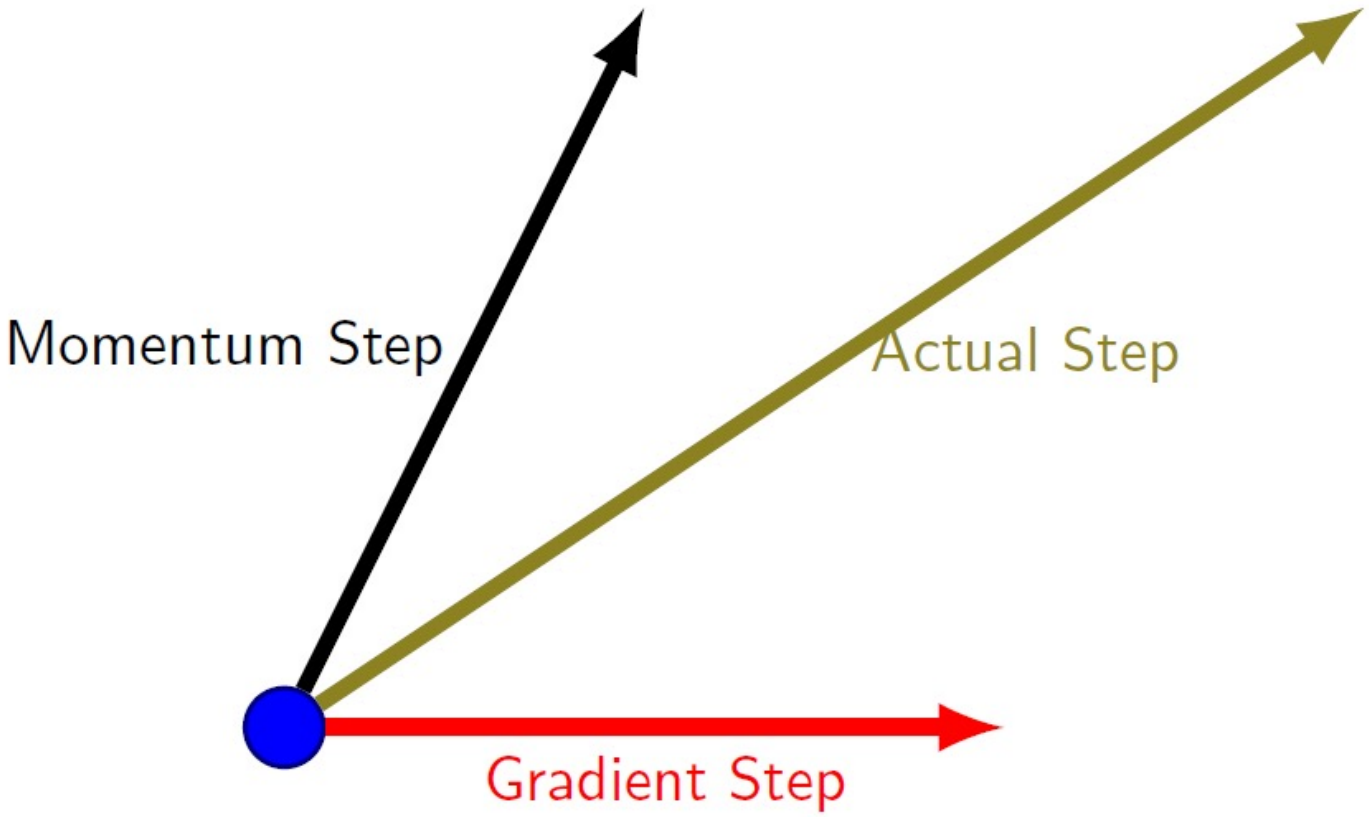
Momentum

- Let's look at the velocity term:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left(L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$$

- The velocity accumulates the previous gradients
- What is the role of α ?
  - If α is larger than ϵ the current update is more affected by the previous gradients
  - Usually values for α are set high $\approx$ 0.8, 0.9

Momentum



Momentum: Step Sizes

- In SGD, the step size was the norm of the gradient scaled by the learning rate $\epsilon\|\mathbf{g}\|$.
- While using momentum, the step size will also depend on the norm and alignment of a sequence of gradients
- For example, if at each step we observed g, the step size would be :

$$\epsilon\frac{\|\mathbf{g}\|}{1-\alpha}$$

- If $\alpha = 0.9 \implies$multiply the maximum speed by 10 relative to the current gradient direction

Momentum

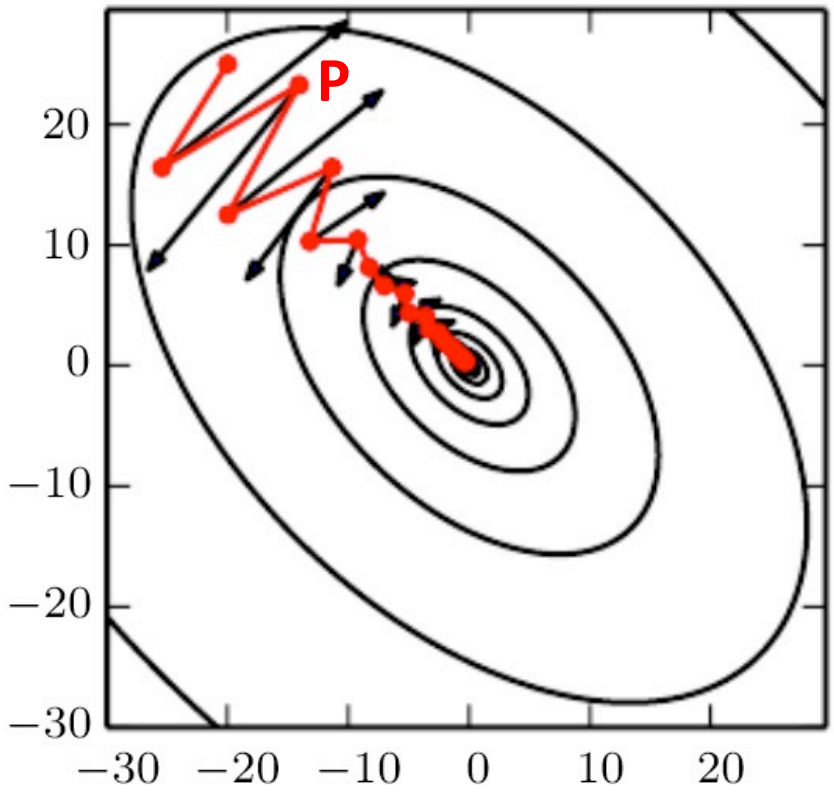


Illustration of how momentum traverses such an error surface better compared to Gradient Descent

SGD with Momentum

Algorithm 2 Stochastic Gradient Descent with Momentum

Require: Learning rate ϵ_k

Require: Momentum Parameter α

Require: Initial Parameter θ

Require: Initial Velocity v

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

3: Compute gradient estimate:

4: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

5: Compute the velocity update:

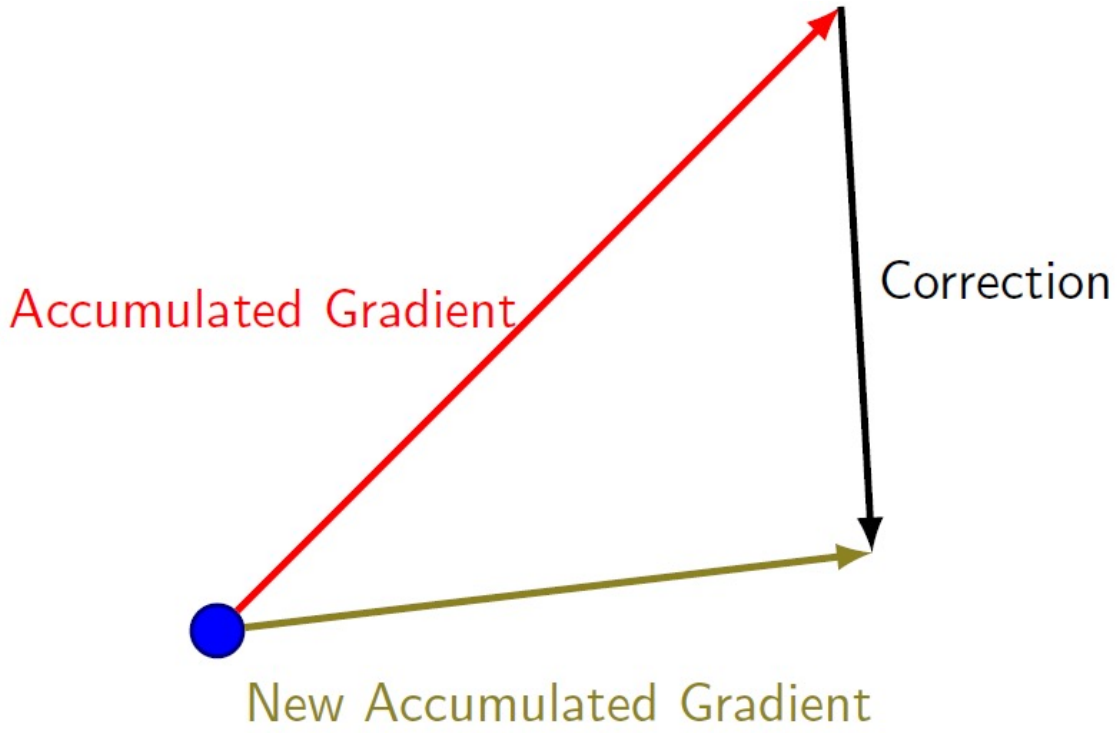
6: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\hat{\mathbf{g}}$

7: Apply Update: $\theta \leftarrow \theta + \mathbf{v}$

8: **end while**

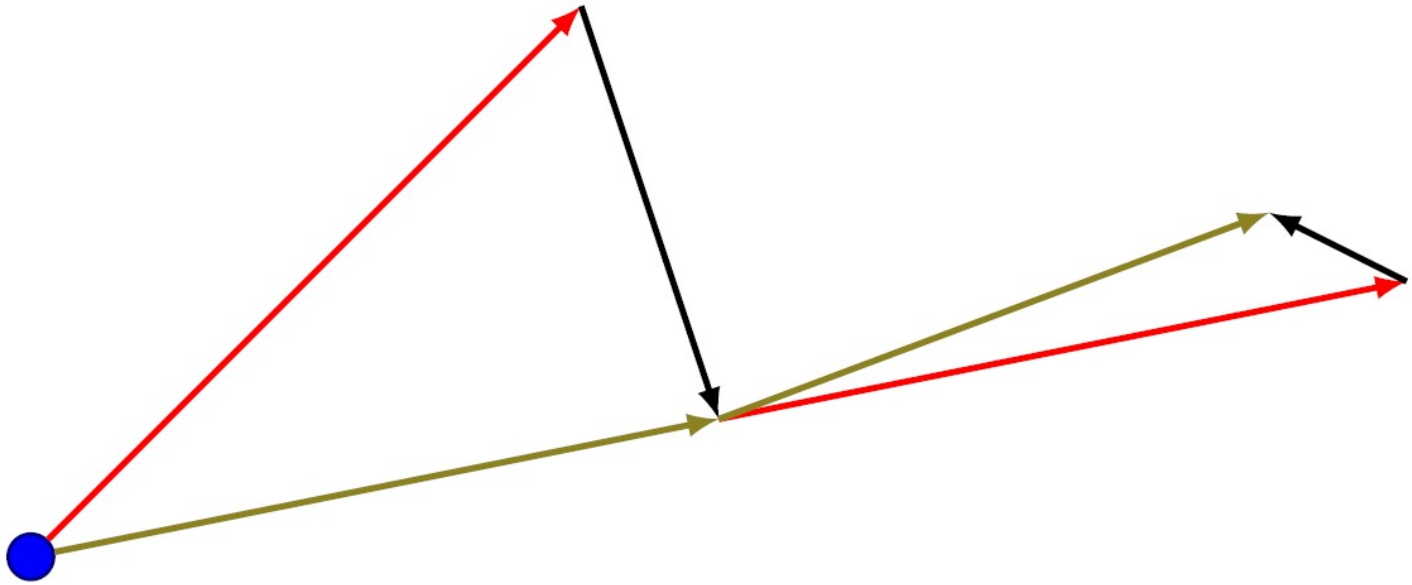
Nesterov Momentum

- Another approach: First take a step in the direction of the accumulated gradient
- Then calculate the gradient and make a correction



Nesterov Momentum

Next Step



Nesterov Momentum

- Recall the velocity term in the Momentum method:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left(L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$$

- Nesterov Momentum:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left(L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right)$$

- Update: $\theta \leftarrow \theta + \mathbf{v}$

SGD with Nesterov Momentum

Algorithm 3 SGD with Nesterov Momentum

Require: Learning rate ϵ

Require: Momentum Parameter α

Require: Initial Parameter θ

Require: Initial Velocity v

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

3: Update parameters: $\tilde{\theta} \leftarrow \theta + \alpha \mathbf{v}$

4: Compute gradient estimate:

5: $\hat{\mathbf{g}} \leftarrow +\nabla_{\tilde{\theta}} L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$

6: Compute the velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$

7: Apply Update: $\theta \leftarrow \theta + \mathbf{v}$

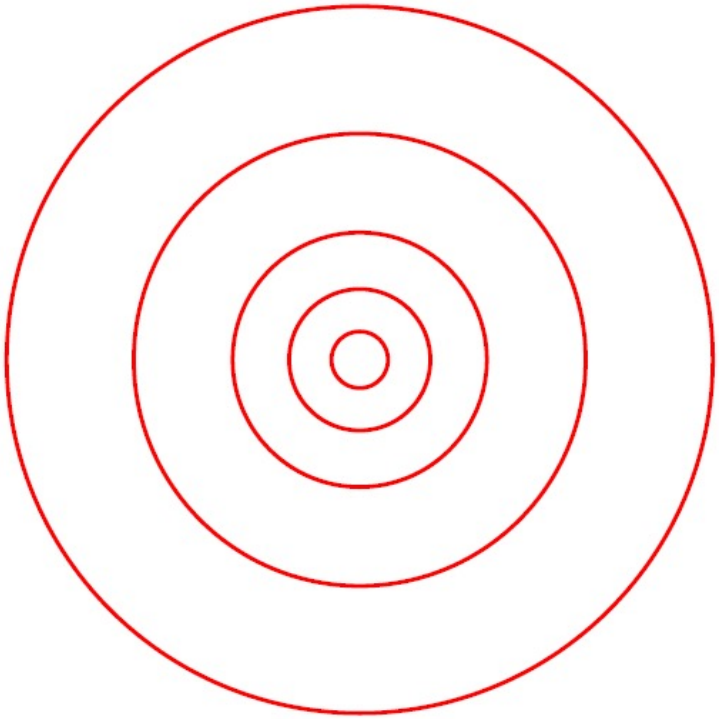
8: **end while**

Adaptive Learning Rate Methods

Motivation

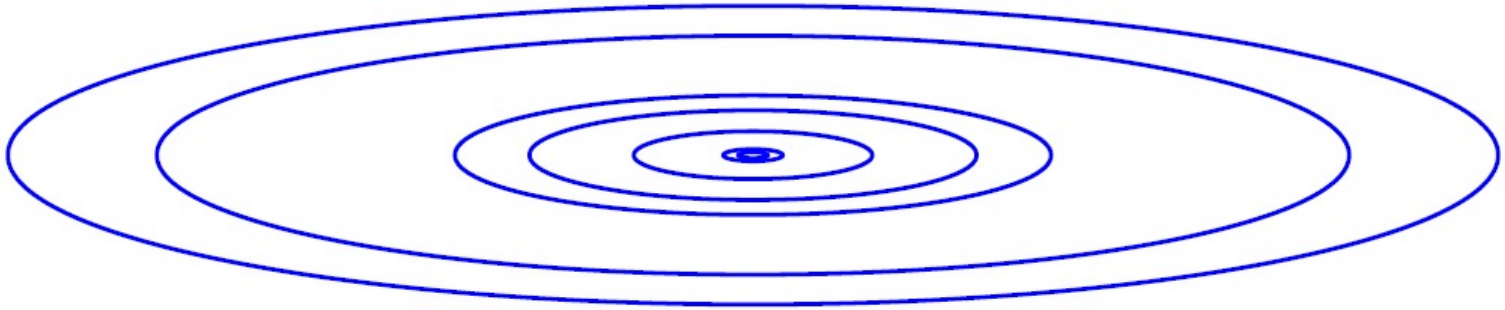
- Till now we assign the same learning rate to all parameters
- If the parameters vary in importance and frequency, why is this a good idea?
- It's probably not!

Motivation



Nice (all parameters are equally important)

Motivation



Harder!

Methods

- AdaGrad
- RMSProp
- Adam

AdaGrad

- Idea: Downscale a model parameter by square-root of sum of squares of all its historical values
- Parameters that have large partial derivative of the loss - learning rates for them are rapidly declined
- While parameters with small partial derivatives have a relatively small decrease in their learning rate

AdaGrad

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.

end while

RMSProp

- AdaGrad is good when the objective is convex.
- AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure.
- We can adapt it to perform better in non-convex settings by accumulating an exponentially decaying average of the gradient
- Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.

RMSProp

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.

end while

RMSProp with Nesterov

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

Initialize accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

Nesterov Momentum: $$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left(L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right)$$

Adam

- We could have used RMSProp with momentum
- The use of momentum in combination with rescaling does not have a clear theoretical motivation.

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}.$$

- Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments

Adam: Adaptive Moments

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})

Require: Initial parameters $\boldsymbol{\theta}$

Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

$t \leftarrow t + 1$

Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$

Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$

Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Conclusion

SGD: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

Momentum: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \mathbf{v}$

Nesterov: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left(L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right)$ then $\theta \leftarrow \theta + \mathbf{v}$

AdaGrad: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ then $\Delta\theta - \leftarrow \frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ then $\theta \leftarrow \theta + \Delta\theta$

RMSProp: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$ then $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \Delta\theta$

Adam: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$ then $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ then $\theta \leftarrow \theta + \Delta\theta$

Batch Normalization

- Method to reparameterize a deep network to reduce co-ordination of update across layers
- Can be applied to input layer, or any hidden layer
- Let H be a design matrix having activations in any layer for m examples in the mini-batch

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1k} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{m1} & h_{m2} & h_{m3} & \dots & h_{mk} \end{bmatrix}$$

Batch Normalization

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1k} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{m1} & h_{m2} & h_{m3} & \dots & h_{mk} \end{bmatrix}$$

- Each row represents all the activations in layers for one example
- Idea: Replace H by H' such that:

$$H' = \frac{H - \mu}{\sigma}$$

- μ is mean of each unit and σ the standard deviation

Batch Normalization

- μ is a vector with μ_j the column mean
- σ is a vector with σ_j the column standard deviation
- $H_{i,j}$ is normalized by subtracting μ_j and dividing by σ_j

Batch Normalization

- During training we have:

$$\mu = \frac{1}{m} \sum_j H_{:,j}$$

$$\sigma = \sqrt{\delta + \frac{1}{m} \sum_j (H - \mu)_j^2}$$

- We then operate on H' as before $\Longrightarrow$ we backpropagate through the normalized activations

An Innovation

- Standardizing the output of a unit can limit the expressive power of the neural network
- Solution: Instead of replacing H by H', replace it will $\gamma H' + \beta$
- γ and β are also learned by backpropagation

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

"Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Ioffe and Szegedy 2015

Why is this good?

- The update will never act to only increase the mean and standard deviation of any activation
- Previous approaches added penalties to cost or per layer to encourage units to have standardized outputs
- Batch normalization makes the reparameterization easier
- At test time: Use running averages of μ and σ collected during training, use these for evaluating new input x

Batch Normalization: Conclusion

- Improves gradient flow through the network.
- Allows higher learning rates.
- Reduces the strong dependence on initialization.
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout.

Initialization Strategies

- In convex problems with good ϵ no matter what the initialization, convergence is guaranteed
- In the non-convex regime initialization is much more important
- Some parameter initialization can be unstable, not converge
- Neural Networks are not well understood to have principled, mathematically nice initialization strategies
- What is known: Initialization should break symmetry
- What is known: Scale of weights is important
- Most initialization strategies are based on intuitions and heuristics

Some Heuristics

- For a fully connected layer with m inputs and n outputs, sample:

$$W_{ij} \sim U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$$

- Xavier Initialization: Sample

$$W_{ij} \sim U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}})$$

- Xavier initialization is derived considering that the network consists of matrix multiplications with no nonlinearities
- Works well in practice!

More Heuristics

- Saxe et al. 2013, recommend initializing to random orthogonal matrices, with a carefully chosen gain g that accounts for non-linearities
- If g could be divined, it could solve the vanishing and exploding gradients problem (more later)
- The idea of choosing g and initializing weights accordingly is that we want norm of activations to increase, and pass back strong gradients
- Martens 2010, suggested an initialization that was sparse: Each unit could only receive k non-zero weights
- Motivation: It is a bad idea to have all initial weights to have the same standard deviation $\frac{1}{\sqrt{m}}$

Outline

1. Course Review
2. Regularization
3. Optimization
4. Project Grouping

Project Logistics

- Final grade = 50% final exam + 25% course project + 25% homework (paper reading)
- All projects must be finished in a team and each team is composed of 5-10 members
- The group project contains a technical report and project presentation
- The course project asks for volunteer presentation, otherwise it will be decided by random sampling

Project Logistics

- The project presentation will be held in the last course. Each team selects one student to report their work and the reporting time is 5 to 10 minutes.

- The technical report must contain responsibility and workload of each member in addition to regular project description. At the same time, the final technical report should be attached with source code which can be reproduced as they present in the technical report.

- The technical report should be written in English with no page limitation and the latex template can be downloaded from http://cvpr2021.thecvf.com/node/33#submission-guidelines.

- Considering the difficulty degree of different tasks, we will give appropriate scores in the final grade.

Project Logistics

Projects will be evaluated based on:

1. The technical quality of the work. (I.e., Does the technical material make sense? Are the things tried reasonable? Are the proposed algorithms or applications clever and interesting? Do the authors convey novel insight about the problem and/or algorithms?)

2. The completeness and novelty of the work, and the clarity of the write-up. (Spare enough time for the write-up since it may be harder than you imagine)

Project Logistics

Your final report is expected to be a 4~8 page report. You should submit both an electronic and a hardcopy version for your final report. It should roughly have the following format:

- Introduction - Motivation
- Problem definition
- Proposed method
  - Intuition - why should it be better than other methods?
  - Description of its algorithms
- Experiments
  - Description of your testbed; list of questions your experiments are designed to answer
  - Details of the experiments; observations
- Conclusions

Candidate Projects

1. **Human Action Recognition**
2. **Image Categorization**
3. **Multi-modal Social Media**
4. **Image Segmentation**
5. **Face Recognition**
6. **Text Classification**
7. **Question Answering**
8. **Image Denosing/Super-resolution**
9. **Image Retrieval**
10. **Tracking**
11. **……**

Note that not all topics are equally difficult. But we'll take this into consideration when evaluating each team's performance and make the assessment as fair as possible.

Acknowledgement

Some of the materials in these slides are drawn inspiration from:

- Shubhendu Trivedi and Risi Kondor, University of Chicago, Deep Learning Course

- Hung-yi Lee, National Taiwan University, Machine Learning and having it Deep and Structured course

- Xiaogang Wang, The Chinese University of Hong Kong, Deep Learning Course

- Fei-Fei Li, Standord University, CS231n Convolutional Neural Networks for Visual Recognition course

Next time

- Deep Generative Models

Questions?

Thank You !