CSC413/2516 Lecture 6: Optimization & Generalization

Jimmy Ba and Bo Wang

a.k.a. Two Bags of Tricks!
Logistics

Some administrative stuff:

- Tuesday’s OH will be changed to 2-3pm!
- Midterm marks: 86% (1.43) Congratulations!
- Final Project Proposal is due Mar 4th
- We will return to in-person teaching after the reading week.
Deep Learning: Alchemy or Science?

AI researchers allege that machine learning is alchemy

By Matthew Hutson | May 3, 2018, 11:15 AM

Ali Rahimi, a researcher in artificial intelligence (AI) at Google in San Francisco, California, took a swipe at his field last December—and received a 40-second ovation for it. Speaking at an AI conference, Rahimi charged that machine learning algorithms, in which computers learn through trial and error, have become a form of “alchemy.” Researchers, he said, do not know why some

Alchemy (ˈāl-kə-mē) noun

the medieval forerunner of chemistry, based on the supposed transformation of matter. It was concerned particularly with attempts to convert base metals into gold or to find a universal elixir.

“Occult sciences, such as alchemy and astrology”
Overview: Gradient Descent

- Goal: we want to minimize a specific objective function (cost or loss):

\[ \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{J}^{(i)}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y(x^{(i)}, \theta), t^{(i)}). \]

- Step 1: Calculate Gradient (by linearity)

\[ \nabla \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{J}^{(i)}(\theta). \]

- Step 2: Update the parameters:

\[ \theta = \theta - \alpha \nabla \mathcal{J}(\theta) \]

Source: https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c
Overview: Learning Rate

- The learning rate $\alpha$ is a hyperparameter we need to tune. Here are the things that can go wrong in batch mode:

  - $\alpha$ too small: slow progress
  - $\alpha$ too large: oscillations
  - $\alpha$ much too large: instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try 0.1, 0.03, 0.01, ...).
Overview: Training Curves

- To diagnose optimization problems, it’s useful to look at training curves: plot the training cost as a function of iteration.

- **Gotcha:** use a fixed subset of the training data to monitor the training error. Evaluating on a different batch (e.g. the current one) in each iteration adds a *lot* of noise to the curve!

- **Gotcha:** it’s very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can’t guarantee convergence.
Overview: Limitations of Gradient Descent

- So far, the cost function $\mathcal{J}$ has been the average loss over the training examples:

$$
\mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{J}^{(i)}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y(x^{(i)}, \theta), t^{(i)}).
$$

- By linearity,

$$
\nabla \mathcal{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{J}^{(i)}(\theta).
$$

- Computing the gradient requires summing over all of the training examples. This is known as batch training.

- Batch training is impractical if you have a large dataset (e.g. millions of training examples)!
Stochastic Gradient Descent

- **Stochastic gradient descent (SGD):** update the parameters based on the gradient for a single training example:

\[ \theta \leftarrow \theta - \alpha \nabla \mathcal{J}^{(i)}(\theta) \]

- SGD can make significant progress before it has even looked at all the data!

- Mathematical justification: if you sample a training example at random, the stochastic gradient is an **unbiased estimate** of the batch gradient:

\[ \mathbb{E}_i \left[ \nabla \mathcal{J}^{(i)}(\theta) \right] = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{J}^{(i)}(\theta) = \nabla \mathcal{J}(\theta). \]
Batch gradient descent moves directly downhill. SGD takes steps in a noisy direction, but moves downhill on average.
Trick 1.1: Mini-batch Gradient Descent

- **Problem:** if we only look at one training example at a time, we can’t exploit efficient vectorized operations.

- **Compromise approach:** compute the gradients on a medium-sized set of training examples, called a mini-batch.

- Each entire pass over the dataset is called an epoch.

- Stochastic gradients computed on larger mini-batches have smaller variance:

  \[
  \text{Var} \left[ \frac{1}{S} \sum_{i=1}^{S} \frac{\partial L^{(i)}}{\partial \theta_j} \right] = \frac{1}{S^2} \text{Var} \left[ \sum_{i=1}^{S} \frac{\partial L^{(i)}}{\partial \theta_j} \right] = \frac{1}{S} \text{Var} \left[ \frac{\partial L^{(i)}}{\partial \theta_j} \right]
  \]

- The mini-batch size $S$ is a hyperparameter.
Trick 1.1: Mini-batch Gradient Descent

Let’s talk in codes:

```python
for t in range(1, num_iters):
    batch = get_batch()
    loss = compute_loss(batch, w)
    dw = compute_gradient(loss)
    w := alpha * dw
```
Trick 1.2: Batch Size

- The mini-batch size $S$ is a hyperparameter that needs to be set.
  - **Large batches**: converge in fewer weight updates because each stochastic gradient is less noisy.
  - **Small batches**: perform more weight updates per second because each one requires less computation.
Trick 1.2: Batch Size

- The mini-batch size $S$ is a hyperparameter that needs to be set.
  - **Large batches**: converge in fewer weight updates because each stochastic gradient is less noisy.
  - **Small batches**: perform more weight updates per second because each one requires less computation.

- **Claim**: If the wall-clock time were proportional to the number of FLOPs, then $S = 1$ would be optimal.
  - 100 updates with $S = 1$ requires the same FLOP count as 1 update with $S = 100$.
  - Rewrite minibatch gradient descent as a for-loop:

    $S = 1$
    \[
    \text{For } k = 1, \ldots, 100:
    \]
    \[
    \theta_k \leftarrow \theta_{k-1} - \alpha \nabla J^{(k)}(\theta_{k-1})
    \]

    $S = 100$
    \[
    \text{For } k = 1, \ldots, 100:
    \]
    \[
    \theta_k \leftarrow \theta_{k-1} - \frac{\alpha}{100} \nabla J^{(k)}(\theta_0)
    \]

- All else being equal, you’d prefer to compute the gradient at a fresher value of $\theta$. So $S = 1$ is better.
Batch Size

- The reason we don’t use $S = 1$ is that larger batches can take advantage of fast matrix operations and parallelism.
- **Small batches:** An update with $S = 10$ isn’t much more expensive than an update with $S = 1$.
- **Large batches:** Once $S$ is large enough to saturate the hardware efficiencies, the cost becomes linear in $S$.
- Cartoon figure, not drawn to scale:

  ![Graphs showing time per weight update and training examples per second vs. batch size for CPU and GPU.](image)

- Since GPUs afford more parallelism, they saturate at a larger batch size. Hence, GPUs tend to favor larger batch sizes.
Batch Size

- The convergence benefits of larger batches also see diminishing returns.
- **Small batches**: large gradient noise, so large benefit from increased batch size
- **Large batches**: SGD approximates the batch gradient descent update, so no further benefit from variance reduction.

- **Right**: # iterations to reach target validation error as a function of batch size. (Shallue et al., 2018)
Batch Size is a slider on the learning process.
- Small batch size gives a learning process that converges quickly at the cost of noise in the training process.
- Large batch size gives a learning process that converges slowly with accurate estimates of the error gradient.

Tips in practice
- A set of good default values to try: 32, 64, 128, 256...
- It is a good idea to review learning curves of model validation error against training time with different batch sizes when tuning the batch size.
- Tune batch size and learning rate after tuning all other hyper-parameters.
- Use better GPUs if you can.
Trick 1.3: Initialization

- If two hidden units have exactly the same bias and exactly the same incoming and outgoing weights, they will always get exactly the same gradient.
  - So they can never learn to be different features.
  - We break symmetry by initializing the weights to have small random values.

- If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.
  - We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to sqrt(fan-in).

Source: Geoffrey Hinton
When activation function is linear (or close to linear):

\[ W = \frac{\text{np.random.randn(fan\_in, fan\_out)}}{\sqrt{\text{fan\_in}}} \]  

“Xavier initialization”  
[Glorot et al., 2010]

When activation function is Relu:

\[ W = \frac{\text{np.random.randn(fan\_in, fan\_out)}}{\sqrt{\text{fan\_in}/2}} \]  

He et al., 2015  
(note additional /2)

A better idea (if possible): Start with a pretrained model!
Proper initialization is an active area of research...

*Understanding the difficulty of training deep feedforward neural networks* by Glorot and Bengio, 2010

*Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* by Saxe et al, 2013

*Random walk initialization for training very deep feedforward networks* by Sussillo and Abbott, 2014

*Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* by He et al., 2015

*Data-dependent Initializations of Convolutional Neural Networks* by Krähenbühl et al., 2015

*All you need is a good init*, Mishkin and Matas, 2015
Trick 1.4: Learning Rate

- In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.

- Typical strategy:
  - Use a large learning rate early in training so you can get close to the optimum
  - Gradually decay the learning rate to reduce the fluctuations
Trick 1.4: Learning Rate

- Guess an initial learning rate.
  - If the error keeps getting worse or oscillates wildly, reduce the learning rate.
  - If the error is falling fairly consistently but slowly, increase the learning rate.
- Write a simple program to automate this way of adjusting the learning rate.
- Towards the end of mini-batch learning it nearly always helps to turn down the learning rate.
  - This removes fluctuations in the final weights caused by the variations between mini-batches.
- Turn down the learning rate when the error stops decreasing.

Source: Geoffrey Hinton
Warning: by reducing the learning rate, you reduce the fluctuations, which can appear to make the loss drop suddenly. But this can come at the expense of long-run performance.
Tip in practice: Typically, a grid search involves picking values approximately on a logarithmic scale, e.g., a learning rate taken within the set {.1, .01, 10^{-3}, 10^{-4}, 10^{-5}}.

=> Learning rate decay over time!

- **step decay:**
  e.g. decay learning rate by half every few epochs.

- **exponential decay:**
  \[ \alpha = \alpha_0 e^{-kt} \]

- **1/t decay:**
  \[ \alpha = \alpha_0 / (1 + kt) \]
The problem with SGD:

Suppose loss function is steep vertically but shallow horizontally:

Q: What is the trajectory along which we converge towards the minimum with SGD?
Trick 1.5: SGD with Momentum

The problem with SGD:
Suppose loss function is steep vertically but shallow horizontally:

Q: What is the trajectory along which we converge towards the minimum with SGD?
The problem with SGD:

Suppose loss function is steep vertically but shallow horizontally:

Q: What is the trajectory along which we converge towards the minimum with SGD? very slow progress along flat direction, jitter along steep one
Trick 1.5: SGD with Momentum

The intuition behind the momentum method

Imagine a ball on the error surface. The location of the ball in the horizontal plane represents the weight vector.

- The ball starts off by following the gradient, but once it has velocity, it no longer does steepest descent.
- Its momentum makes it keep going in the previous direction.

- It damps oscillations in directions of high curvature by combining gradients with opposite signs.
- It builds up speed in directions with a gentle but consistent gradient.

Source: Geoffrey Hinton
Trick 1.5: SGD with Momentum

$W_t$
Trick 1.5: SGD with Momentum

$W_t$ \rightarrow \nabla L(w_t)$
Trick 1.5: SGD with Momentum

Source: Kosta Derpanis
Trick 1.5: SGD with Momentum

Combine gradient and velocity to get new weights.
Trick 1.5: SGD with Momentum

Source: Kosta Derpanis

velocity

combine gradient and velocity to get new weights

$w_t$  $\nabla L(w_t)$  $w_{t+1}$
Trick 1.5: SGD with Momentum

Let’s talk in codes:

```python
vw = 0
beta1 = 0.9

for t in range(1, num_iters):
    batch = get_batch()
    loss = compute_loss(batch, w)

    dw = compute_gradient(loss)
    vw = beta1 * vw + (1 - beta1) * dw

    w -= alpha * vw
```
Trick 1.6: RMSProp (optional)

**rmsprop: A mini-batch version of rprop**

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight
  \[
  \text{MeanSquare}(w, t) = 0.9 \text{MeanSquare}(w, t-1) + 0.1 \left( \frac{\partial E}{\partial w}(t) \right)^2
  \]
- Dividing the gradient by \( \sqrt{\text{MeanSquare}(w, t)} \) makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton’s Coursera class, lecture 6

Cited by several papers as:

Let's talk in codes:

```python
moment2 = 0
beta2 = 0.9
c = 10e-8

for t in range(1, num_iters):
    batch = get_batch()
    loss = compute_loss(batch, w)

    dw = compute_gradient(loss)
    moment2 = beta2*moment2 + (1 - beta2)*dw*dw

    w -= alpha * dw / (sqrt(moment2) + c)
```
Trick 1.7: Adam = Momentum + RMSProp (optional)

**ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION**

*Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

*Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

**ABSTRACT**

We introduce *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with model misspecification.
Trick 1.7: Adam = Momentum + RMSProp (optional)

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

ABSTRACT

We introduce Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very large minibatches.
Trick 1.7: Adam = Momentum + RMSProp (optional)

**ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION**

**Diederik P. Kingma**
University of Amsterdam, OpenAI
dpkingma@openai.com

**Jimmy Lei Ba**
University of Toronto
jimmy@psi.utoronto.ca

**ABSTRACT**

We introduce Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very large training sets. The method can be understood and extended in a variety of ways.  \[ \text{Adam combines Momentum and RMSProp} \]

- May not converge to optimal solution
- Works extremely well in practice! (60k+ citations)
Adam = Momentum + RMSProp (optional)

Let’s talk in codes:

```python
moment1 = 0
moment2 = 0
beta1 = 0.9
beta2 = 0.999
c = 10e-8

for t in range(1, num_iters):
    batch = get_batch()
    loss = compute_loss(batch, w)

    dw = compute_gradient(loss)
    momentum1 = beta1*moment1 + (1 - beta1)*dw
    moment2 = beta2*moment2 + (1 - beta2)*dw*dw

    w -= alpha * moment1 / (sqrt(moment2) + c)
```
Adam = Momentum + RMSProp (optional)

Let’s talk in codes:

```python
moment1 = 0
moment2 = 0
beta1 = 0.9
beta2 = 0.999
c = 10e-8

for t in range(1, num_iters):
    batch = get_batch()
    loss = compute_loss(batch, w)

    dw = compute_gradient(loss)

    *momentum*
    moment1 = beta1*moment1 + (1 - beta1)*dw

    *RMSProp*
    moment2 = beta2*moment2 + (1 - beta2)*dw*dw

    *Bias Correction*
    unbiased1 = moment1 / (1 - beta1**t)
    unbiased2 = moment2 / (1 - beta2**t)

    w -= alpha * unbiased1 / (sqrt(unbiased2) + c)
```
A toy example
Take-away: Babysit the learning process

<table>
<thead>
<tr>
<th>Problem</th>
<th>Diagnostics</th>
<th>Workarounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>incorrect gradients</td>
<td>finite differences</td>
<td>fix them, or use autodiff</td>
</tr>
<tr>
<td>local optima</td>
<td>(hard)</td>
<td>random restarts</td>
</tr>
<tr>
<td>slow progress</td>
<td>slow, linear training curve</td>
<td>increase $\alpha$; momentum</td>
</tr>
<tr>
<td>instability</td>
<td>cost increases</td>
<td>decrease $\alpha$</td>
</tr>
<tr>
<td>oscillations</td>
<td>fluctuations in training curve</td>
<td>decrease $\alpha$; momentum</td>
</tr>
<tr>
<td>fluctuations</td>
<td>fluctuations in training curve</td>
<td>decay $\alpha$; iterate averaging</td>
</tr>
<tr>
<td>dead/saturated units</td>
<td>activation histograms</td>
<td>initial scale of $\mathbf{W}$; ReLU</td>
</tr>
<tr>
<td>ill-conditioning</td>
<td>(hard)</td>
<td>normalization; momentum; Adam; second-order opt.</td>
</tr>
</tbody>
</table>
Take-away: Good practice in learning process

- Tune the batch size using grid search by monitoring the training and validation curves
- Tune the initial learning rate using grid search by monitoring the training and validation curves
- Use adaptive learning rate decay
- Choose a good initialization
- Adam and SGD with momentum are good default optimization methods
Let’s take a break!

- We’ve focused so far on how to *optimize* neural nets — how to get them to make good predictions on the training set.
- How do we make sure they generalize to data they haven’t seen before?
- Even though the topic is well studied, it’s still poorly understood.
Generalization

Recall: overfitting and underfitting

![Graphs showing overfitting and underfitting](image)

We’d like to minimize the generalization error, i.e. error on novel examples.
Training and test error as a function of # training examples and # parameters:
Our Second Bag of Tricks

- How can we train a model that’s complex enough to model the structure in the data, but prevent it from overfitting? I.e., how to achieve low bias and low variance?
- Our bag of tricks
  - data augmentation
  - reduce the number of parameters
  - weight decay
  - early stopping
  - ensembles (combine predictions of different models)
  - stochastic regularization (e.g. dropout)
- The best-performing models on most benchmarks use some or all of these tricks.
Trick 2.1: Data Augmentation

- The best way to improve generalization is to collect more data!
- Suppose we already have all the data we’re willing to collect. We can augment the training data by transforming the examples. This is called data augmentation.
- Examples (for visual recognition)
  - translation
  - horizontal or vertical flip
  - rotation
  - smooth warping
  - noise (e.g. flip random pixels)
- Only warp the training, not the test, examples.
- The choice of transformations depends on the task. (E.g. horizontal flip for object recognition, but not handwritten digit recognition.)
Typical Image Augmentation

Source: Sourav Kumar
Trick 2.1: Data Augmentation

Time series data augmentation

(a) Original
(b) Jittering
(c) Scaling
(d) Magnitude Warping
(e) Rotation
(f) Permutation
(g) Window Slice
(h) Time Warping
(i) Window Warping
This article will focus on summarizing data augmentation techniques in NLP.

This write-up will focus on summarizing data augmentation methods in NLP.

Source: Shahul ES
This article will focus on summarizing data augmentation techniques in NLP.

Random Insertion

This article will focus on write-up summarizing data augmentation techniques in NLP methods.

Source: Shahul ES
This article will focus on summarizing data augmentation techniques in NLP.

Random Swap

This techniques will focus on summarizing data augmentation article in NLP.

Source: Shahul ES
Data Augmentation for NLP

I have no time

I do not have time

Back-Translation

Source: Shahul ES
Trick 2.1: Data Augmentation

Caution!

Label: 6

Data Augmentation
Rotate 180°

New Label: 9
Ground truth label: 6
Trick 2.2: Reducing the Number of Parameters

- Can reduce the number of layers or the number of parameters per layer.
- Adding a linear **bottleneck layer** is another way to reduce the number of parameters:

![Diagram showing two network architectures](image)

- The first network is strictly more expressive than the second (i.e. it can represent a strictly larger class of functions). (Why?)
- Remember how linear layers don’t make a network more expressive? They might still improve generalization.
Trick 2.2: Reducing the Number of Parameters

An example of bottle-neck structure

Jason Chen’s Blog
Trick 2.3: Weight Decay

- We’ve already seen that we can regularize a network by penalizing large weight values, thereby encouraging the weights to be small in magnitude.

\[ J_{\text{reg}} = J + \lambda R = J + \frac{\lambda}{2} \sum_j w_j^2 \]

- We saw that the gradient descent update can be interpreted as weight decay:

\[
\mathbf{w} \leftarrow \mathbf{w} - \alpha \left( \frac{\partial J}{\partial \mathbf{w}} + \lambda \frac{\partial R}{\partial \mathbf{w}} \right)
\]

\[
= \mathbf{w} - \alpha \left( \frac{\partial J}{\partial \mathbf{w}} + \lambda \mathbf{w} \right)
\]

\[
= (1 - \alpha \lambda) \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}}
\]
Weight Decay

Why we want weights to be small:

\[ y = 0.1x^5 + 0.2x^4 + 0.75x^3 - x^2 - 2x + 2 \]

\[ y = -7.2x^5 + 10.4x^4 + 24.5x^3 - 37.9x^2 - 3.6x + 12 \]

The red polynomial overfits. Notice it has really large coefficients.
Weight Decay

Why we want weights to be small:

- Suppose inputs $x_1$ and $x_2$ are nearly identical. The following two networks make nearly the same predictions:

![Diagram showing two networks with nearly identical predictions.]

- But the second network might make weird predictions if the test distribution is slightly different (e.g. $x_1$ and $x_2$ match less closely).
Weight Decay

- The geometric picture:
Weight Decay

- There are other kinds of regularizers which encourage weights to be small, e.g. sum of the absolute values.
- These alternative penalties are commonly used in other areas of machine learning, but less commonly for neural nets.
- Regularizers differ by how strongly they prioritize making weights exactly zero, vs. not being very large.

---

— Hinton, Coursera lectures

— Bishop, *Pattern Recognition and Machine Learning*
Trick 2.4: Early Stopping

- We don’t always want to find a global (or even local) optimum of our cost function. It may be advantageous to stop training early.

- Early stopping: monitor performance on a validation set, stop training when the validation error starts going up.
Early Stopping

- A slight catch: validation error fluctuates because of stochasticity in the updates.

![Graph showing training vs. validation error over epochs]

- Determining when the validation error has actually leveled off can be tricky.
Early Stopping

- Why does early stopping work?
  - Weights start out small, so it takes time for them to grow large. Therefore, it has a similar effect to weight decay.
  - If you are using sigmoidal units, and the weights start out small, then the inputs to the activation functions take only a small range of values.
    - Therefore, the network starts out approximately linear, and gradually becomes more nonlinear (and hence more powerful).
Trick 2.5: Ensembles

- If a loss function is convex (with respect to the predictions), you have a bunch of predictions, and you don’t know which one is best, you are always better off averaging them.

\[ \mathcal{L}(\lambda_1 y_1 + \cdots + \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \cdots + \lambda_N \mathcal{L}(y_N, t) \quad \text{for } \lambda_i \geq 0, \sum \lambda_i = 1 \]

- This is true no matter where they came from (trained neural net, random guessing, etc.). Note that only the loss function needs to be convex, not the optimization problem.

- Examples: squared error, cross-entropy, hinge loss

- If you have multiple candidate models and don’t know which one is the best, maybe you should just average their predictions on the test data. The set of models is called an ensemble.

- Averaging often helps even when the loss is nonconvex (e.g. 0–1 loss).
Ensembles

- Some examples of ensembles:
  - Train networks starting from different random initializations. But this might not give enough diversity to be useful.
  - Train networks on different subsets of the training data. This is called bagging.
  - Train networks with different architectures or hyperparameters, or even use other algorithms which aren’t neural nets.

- Ensembles can improve generalization quite a bit, and the winning systems for most machine learning benchmarks are ensembles.

  - But they are expensive, and the predictions can be hard to interpret.
Trick 2.6: Stochastic Regularization

- For a network to overfit, its computations need to be really precise. This suggests regularizing them by injecting noise into the computations, a strategy known as **stochastic regularization**.

- **Dropout** is a stochastic regularizer which randomly deactivates a subset of the units (i.e. sets their activations to zero).

\[
  h_j = \begin{cases} 
    \phi(z_j) & \text{with probability } 1 - \rho \\
    0 & \text{with probability } \rho,
  \end{cases}
\]

where \( \rho \) is a hyperparameter.

- Equivalently,

\[
  h_j = m_j \cdot \phi(z_j),
\]

where \( m_j \) is a Bernoulli random variable, independent for each hidden unit.

- Backprop rule:

\[
  \overline{z}_j = \overline{h}_j \cdot m_j \cdot \phi'(z_j)
\]
Trick 2.6: Stochastic Regularization

Source: Peter Skalski
Stochastic Regularization

- Dropout can be seen as training an ensemble of $2^D$ different architectures with shared weights (where $D$ is the number of units):

---

“Goodfellow et al., *Deep Learning*”
Dropout

Dropout at test time:

- Most principled thing to do: run the network lots of times independently with different dropout masks, and average the predictions.
  - Individual predictions are stochastic and may have high variance, but the averaging fixes this.
- In practice: don’t do dropout at test time, but multiply the weights by $1 - \rho$
  - Since the weights are on $1 - \rho$ fraction of the time, this matches their expectation.
Consider a linear regression, $y^{(i)} = \sum_j w_j x_j^{(i)}$. The inputs are dropped out half of the time: $\tilde{y}^{(i)} = 2 \sum_j m_j^{(i)} w_j x_j^{(i)}$, $m \sim \text{Bern}(0.5)$. $\mathbb{E}_m[\tilde{y}^{(i)}] = y^{(i)}$.

$$\mathbb{E}_m[J] = \frac{1}{2N} \sum_{i=1}^{N} \mathbb{E}_m[(\tilde{y}^{(i)} - t^{(i)})^2]$$
Dropout as an Adaptive Weight Decay

Consider a linear regression, \( y^{(i)} = \sum_j w_j x_j^{(i)} \). The inputs are dropped out half of the time: \( \tilde{y}^{(i)} = 2 \sum_j m_j^{(i)} w_j x_j^{(i)} \), \( m \sim \text{Bern}(0.5) \). \( \mathbb{E}_m[\tilde{y}^{(i)}] = y^{(i)} \).

\[
\mathbb{E}_m[\mathcal{J}] = \frac{1}{2N} \sum_{i=1}^{N} \mathbb{E}_m[(\tilde{y}^{(i)} - t^{(i)})^2]
\]

The bias-variance decomposition of the squared error gives:

\[
\mathbb{E}_m[\mathcal{J}] = \frac{1}{2N} \sum_{i=1}^{N} (\mathbb{E}_m[\tilde{y}^{(i)}] - t^{(i)})^2 + \frac{1}{2N} \sum_{i=1}^{N} \text{Var}_m[\tilde{y}^{(i)}]
\]
Tips: \( \text{Var}[m_{j}] = p \ (1-p) \)
Stochastic Regularization

- Dropout can help performance quite a bit, even if you’re already using weight decay.
- Lots of other stochastic regularizers have been proposed:
  - Batch normalization (mentioned last week for its optimization benefits) also introduces stochasticity, thereby acting as a regularizer.
  - The stochasticity in SGD updates has been observed to act as a regularizer, helping generalization.
    - Increasing the mini-batch size may improve training error at the expense of test error!
Take-away: Our Bag of Tricks

- Techniques we just covered:
  - data augmentation
  - reduce the number of parameters
  - weight decay
  - early stopping
  - ensembles (combine predictions of different models)
  - stochastic regularization (e.g. dropout)

- The best-performing models on most benchmarks use some or all of these tricks.
Revisit the Controversy

Deep Learning: Alchemy or Science?

“The engineering artifacts have almost always preceded the theoretical understanding,” said LeCun. “Understanding (theoretical or otherwise) is a good thing. It’s the very purpose of many of us in the NIPS community. But another important goal is inventing new methods, new techniques, and yes, new tricks.”