Deep Learning General Strategy and Tricks

Dr. Kira Radinsky
CTO SalesPredict
Visiting Professor/Scientist Technion

Slides were adapted from lectures by Richard Socher
Overview Today:

• Useful NNet techniques / tips and tricks:
  • Training with Backpropagation – Vectorized
  • Nonlinearities
  • Finite difference gradient check
  • Momentum, AdaGrad
• This is a 3 layer neural network
• 1 hidden-layer neural network
Toy Example: Terminology

Layer 1  Layer 2  Layer 3

Model Input

Model Output
Toy Example: Terminology

Layer 1  Layer 2  Layer 3

$x_1$  $z_1^{(1)}$  $a_1^{(1)}$  $z_1^{(2)}$  $a_1^{(2)}$  $z_1^{(3)}$  $a_1^{(3)}$

$x_2$  $z_2^{(1)}$  $a_2^{(1)}$  $z_2^{(2)}$  $a_2^{(2)}$

$x_3$  $z_3^{(1)}$  $a_3^{(1)}$

$x_4$  $z_4^{(1)}$  $a_4^{(1)}$

Model Input  Activation Units  Model Output
We draw this

\[
z_{1}^{(2)} = W_{11}^{(1)}a_{1}^{(1)} + W_{12}^{(1)}a_{2}^{(1)} + W_{13}^{(1)}a_{3}^{(1)} + W_{14}^{(1)}a_{4}^{(1)}
\]

\[a_{1}^{(2)} \text{ is the } 1^{\text{st}} \text{ activation unit of layer } 2\]

\[a_{1}^{(2)} = \sigma(z_{1}^{(2)})\]

This is actually what’s going on
Forward Pass

\[ z_1^{(1)} = x_1 \]
\[ z_2^{(1)} = x_2 \]
\[ z_3^{(1)} = x_3 \]
\[ z_4^{(1)} = x_4 \]
Forward Pass

\[ a_1^{(1)} = z_1^{(1)} \]
\[ a_2^{(1)} = z_2^{(1)} \]
\[ a_3^{(1)} = z_3^{(1)} \]
\[ a_4^{(1)} = z_4^{(1)} \]
Forward Pass

\[ z_1^{(2)} = W_{11}^{(1)} a_1^{(1)} + W_{12}^{(1)} a_2^{(1)} + W_{13}^{(1)} a_3^{(1)} + W_{14}^{(1)} a_4^{(1)} \]

\[ z_2^{(2)} = W_{21}^{(1)} a_1^{(1)} + W_{22}^{(1)} a_2^{(1)} + W_{23}^{(1)} a_3^{(1)} + W_{24}^{(1)} a_4^{(1)} \]
Forward Pass

\[
\begin{pmatrix}
    z_1^{(2)} \\
    z_2^{(2)}
\end{pmatrix} =
\begin{pmatrix}
    W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} & W_{14}^{(1)} \\
    W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} & W_{24}^{(1)}
\end{pmatrix}
\begin{pmatrix}
    a_1^{(1)} \\
    a_2^{(1)} \\
    a_3^{(1)} \\
    a_4^{(1)}
\end{pmatrix}
\]
Forward Pass

\[
z^{(2)} = W^{(1)}a^{(1)}
\]

Affine transformation
Forward Pass

\[ a^{(2)} = \sigma(z^{(2)}) \]

Point-wise/Element-wise non-linearity
Forward Pass

\[ z^{(3)} = W^{(2)}a^{(2)} \]

Affine transformation
Forward Pass

\[ a^{(3)} = z^{(3)} \]

\[ s = a^{(3)} \]
Let us try to calculate the error gradient wrt $W_{14}^{(1)}$. Thus we want to find:

$$\frac{\partial s}{\partial W_{14}^{(1)}}$$
Let us try to calculate the error gradient wrt $W_{14}^{(1)}$. Thus we want to find:

\[
\begin{align*}
\frac{\partial s}{\partial z_1^{(3)}} &\quad \frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} &\quad \frac{\partial a_1^{(2)}}{\partial z_2^{(2)}} &\quad \frac{\partial z_2^{(2)}}{\partial W_{14}^{(1)}}
\end{align*}
\]
Backpropogation using chain rule

This is simply 1

\[
\begin{bmatrix}
\frac{\partial s}{\partial z_1^{(3)}} & \frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} & \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} & \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}} \\
\end{bmatrix}
\]
Backpropagation using chain rule

\[
\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}} \frac{\partial (W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)})}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}}
\]
Backpropagation using chain rule

\[ W_{11}^{(2)} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}} \]
Backpropagation using chain rule

\[ W_{11}^{(2)} \sigma'(z_1^{(2)}) \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}} \]
Backpropagation using chain rule

\[ W_{11}^{(2)} \sigma' (z_1^{(2)}) \frac{\partial (W_{11}^{(1)} a_1^{(1)} + W_{12}^{(1)} a_2^{(1)} + W_{13}^{(1)} a_3^{(1)} + W_{14}^{(1)} a_4^{(1)})}{\partial W_{14}^{(1)}} \]
Backpropagation using chain rule

\[ W_{11}^{(2)} \sigma' \left( z_1^{(2)} \right) a_4^{(1)} \]

\[ \delta_1^{(2)} \]
We got error gradient wrt $W_{14}^{(1)}$.

Required:
- the signal forwarded by $W_{14}^{(1)} = a^{(1)}_4$
- the error propagating backwards $W_{11}^{(2)}$
- the local gradient $\sigma'(z_1^{(2)})$
We tried to get error gradient wrt $W_{14}^{(1)}$

Required:
- the signal forwarded by $W_{14}^{(1)} = a_4^{(1)}$
- the error propagating backwards $W_{11}^{(2)}$
- the local gradient $\sigma'(z_1^{(2)})$

We can do this for all of $W^{(1)}$:

(as outer product)
Let us define $\delta$

Recall that this is forward pass

This is the backpropagation

$\delta^{(2)}_1$ is the error flowing backwards at the same point where $z^{(2)}_1$ passed forwards. Thus it is simply the gradient of the error wrt $z^{(2)}_1$. 

The chain rule of differentiation just boils down very simple patterns in error backpropagation:

1. An error $x$ flowing backwards passes a neuron by getting amplified by the local gradient.

2. An error $\delta$ that needs to go through an affine transformation distributes itself in the way signal combined in forward pass.
Backpropagation using error vectors
Backpropagation using error vectors

This is $\hat{y} - y$ for softmax
Gradient w.r.t $W^{(2)} = \delta^{(3)}a^{(2)T}$
Backpropagation using error vectors

\[ z^{(1)} \xrightarrow{W^{(1)}} a^{(1)} \xrightarrow{W^{(2)}} z^{(2)} \xrightarrow{\sigma} a^{(2)} \xrightarrow{W^{(2)}} z^{(3)} \xrightarrow{1} s \]

- Reusing the $\delta^{(3)}$ for downstream updates.
- Moving error vector across affine transformation simply requires multiplication with the transpose of forward matrix.
- Notice that the dimensions will line up perfectly too!
Backpropagation using error vectors

\[ \sigma'(z^{(2)}) \odot W^{(2)T} \delta^{(3)} = \delta^{(2)} \]

--Moving error vector across point-wise non-linearity requires point-wise multiplication with local gradient of the non-linearity
Backpropagation using error vectors

\[
\begin{align*}
\sigma(1) &= W^{(1)}a^{(1)} \\
\sigma(2) &= W^{(2)}\sigma(1) \\
s &= 1
\end{align*}
\]

Gradient w.r.t $W^{(1)} = \delta^{(2)}a^{(1)T}$
Backpropagation using error vectors (4-layer network)
Backpropagation using error vectors (4-layer network)
Backpropagation using error vectors (4-layer network)

\[ \text{Grad } W^{(3)} = \delta^{(4)}a^{(3)T} \]
Backpropagation using error vectors (4-layer network)

\[ \delta^{(3)} = \sigma'(z^{(3)}) \odot W^{(3)^T} \delta^{(4)} \]
Backpropagation using error vectors (4-layer network)

\[ \text{Grad } W^{(2)} = \delta^{(3)}a^{(2)T} \]
Backpropagation using error vectors (4-layer network)

\[ \delta^{(2)} = \sigma'(z^{(2)}) \odot W^{(2)^T} \delta^{(3)} \]
Backpropagation using error vectors (4-layer network)

\[
\text{Grad } W^{(1)} = \delta^{(2)}a^{(1)T}
\]

\[
\begin{align*}
W^{(1)} & \rightarrow a^{(1)} = \sigma(z^{(1)}) \\
W^{(2)} & \rightarrow a^{(2)} = \sigma(z^{(2)}) \\
W^{(3)} & \rightarrow a^{(3)} = \sigma(z^{(3)}) \\
\text{Softmax} & \rightarrow y_p = \text{argmax}(z^{(4)})
\end{align*}
\]
Backpropagation using error vectors (4-layer network)

Grad wrt input vector = $W^{(1)T} \delta^{(2)}$
General Strategy for Successful NNets

1. Select network structure appropriate for problem
   1. Structure: Single words, fixed windows, bag of words, recursive vs. recurrent, CNN, sentence based vs. document
   2. Nonlinearity
2. Check for implementation bugs with gradient checks
3. Parameter initialization
4. Optimization tricks
5. Check if the model is powerful enough to overfit
   1. If not, change model structure or make model “larger”
   2. If you can overfit: Regularize
General Strategy for Successful NNets

1. Select network structure appropriate for problem
   1. Structure: Single words, fixed windows, bag of words, recursive vs. recurrent, CNN, sentence based vs. document
   2. Nonlinearity

2. Check for implementation bugs with gradient checks

3. Parameter initialization

4. Optimization tricks

5. Check if the model is powerful enough to overfit
   1. If not, change model structure or make model “larger”
   2. If you can overfit: Regularize
Non-linearities: What’s used

**logistic ("sigmoid")**

\[ f(z) = \frac{1}{1 + \exp(-z)}. \]

\[ f'(z) = f(z)(1 - f(z)) \]

**tanh**

\[ f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} , \]

\[ f'(z) = 1 - f(z)^2 \]

tanh is just a rescaled and shifted sigmoid

tanh often performs well for deep nets

\[ \tanh(z) = 2\text{logistic}(2z) - 1 \]
For many models, tanh is the best!

• In comparison to sigmoid:

• At initialization: values close to 0

• Faster convergence in practice

• Like sigmoid: Nice derivative: \[ f'(z) = 1 - \text{tanh}^2(z) \]
Non-linearities: There are various other choices

**Hard tanh**

\[
\text{HardTanh}(x) = \begin{cases} 
-1 & \text{if } x < -1 \\
 x & \text{if } -1 \leq x \leq 1 \\
 1 & \text{if } x > 1 
\end{cases}
\]

- hard tanh similar but computationally cheaper than tanh and saturates hard.
- Glorot and Bengio, *AISTATS 2011* discuss softsign and rectifier

**Soft sign**

\[
\text{softsign}(z) = \frac{a}{1 + |a|}
\]

**Rectified linear (ReLU)**

\[
\text{rect}(z) = \max(z, 0)
\]
MaxOut Network

A recent type of nonlinearity/network

Goodfellow et al. (2013)

Where \[ f_i(z) = \max_{j \in [1,k]} z_{ij} \]

\[ z_{ij} = x^T W_{ij} + b_{ij} \]

This function too is a universal approximator

State of the art on several image datasets
General Strategy for Successful NNets

1. Select network structure appropriate for problem
   1. Structure: Single words, fixed windows, bag of words, recursive vs. recurrent, CNN, sentence based vs. document
   2. Nonlinearity

2. Check for implementation bugs with gradient checks

3. Parameter initialization

4. Optimization tricks

5. Check if the model is powerful enough to overfit
   1. If not, change model structure or make model “larger”
   2. If you can overfit: Regularize
Gradient Checks are Awesome!

- Allow you to know that there are no bugs in your neural network implementation!
- Steps:
  1. Implement your gradient
  2. Implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon (~$10^{-4}$) and estimate derivatives

\[
 f'(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon} \quad \theta^{(i+)} = \theta + \epsilon \times e_i
\]

3. Compare the two and make sure they are almost the same via relative error!

\[
\frac{|f_{a'} - f_{n'}|}{\max(|f_{a'}|, |f_{n'}|)}
\]
**Use the centered formula.** The formula you may have seen for the finite difference approximation when evaluating the numerical gradient looks as follows:

$$\frac{df(x)}{dx} = \frac{f(x + h) - f(x)}{h} \quad \text{(bad, do not use)}$$

where $h$ is a very small number, in practice approximately $1e-5$ or so. In practice, it turns out that it is much better to use the *centered* difference formula of the form:

$$\frac{df(x)}{dx} = \frac{f(x + h) - f(x - h)}{2h} \quad \text{(use instead)}$$

This requires you to evaluate the loss function twice to check every single dimension of the gradient (so it is about 2 times as expensive), but the gradient approximation turns out to be much more precise. To see this, you can use Taylor expansion of $f(x + h)$ and $f(x - h)$ and verify that the first formula has an error on order of $O(h)$, while the second formula only has error terms on order of $O(h^2)$ (i.e. it is a second order approximation).
A naive implementation of numerical gradient of \( f \) at \( x \)

- \( f \) should be a function that takes a single argument
- \( x \) is the point (numpy array) to evaluate the gradient at

```python
fx = f(x)  # evaluate function value at original point
grad = np.zeros(x.shape)
h = 0.00001

# iterate over all indexes in x
it = np.nditer(x, flags=['multi_index'],
op_flags=['readwrite'])
while not it.finished:

    # evaluate function at \( x+h \)
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h  # increment by \( h \)
    fxh = f(x)  # evaluate \( f(x + h) \)
    x[ix] = old_value  # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h  # the slope
    it.iternext()  # step to next dimension

return grad
```

\[
\frac{\partial J}{\partial \theta} \approx \lim_{h \to 0} \frac{J(\theta + h) - J(\theta - h)}{2h}
\]

\( J(\theta + h) \) and \( J(\theta - h) \) can be evaluated using two forward passes.
Gradient Checks in Practice

- relative error > 1e-2 usually means the gradient is probably wrong
- 1e-2 > relative error > 1e-4 should make you feel uncomfortable
- 1e-4 > relative error is usually okay for objectives with kinks. But if there are no kinks (e.g. use of tanh nonlinearities and softmax), then 1e-4 is too high.
- 1e-7 and less you should be happy.

But - the deeper the network, the higher the relative errors will be....
Gradient Checks in Practice

• Use double precision
• Stick around active range of floating point: "What Every Computer Scientist Should Know About Floating-Point Arithmetic"
• Kinks in the objective
• Use only few datapoints- since loss functions that contain kinks (e.g. due to use of ReLUs or margin losses etc.) will have fewer kinks with fewer datapoints
• Be careful with the step size h. smaller != better => start running into numerical precision problems
Gradient Checks are Awesome!

• If you gradient fails and you don’t know why?
• What now?
• Simplify your model until you have no bug!
• Example: Start from simplest model then go to what you want:
  • Only softmax on fixed input
  • Backprop into word vectors and softmax
  • Add single unit single hidden layer
  • Add multi unit single layer
  • Add bias
  • Add second layer single unit
  • Add two softmax units
General Strategy for Successful NNets

1. Select network structure appropriate for problem
   1. Structure: Single words, fixed windows, bag of words, recursive vs. recurrent, CNN, sentence based vs. document
   2. Nonlinearity
2. Check for implementation bugs with gradient checks
3. Parameter initialization
4. Optimization tricks
5. Check if the model is powerful enough to overfit
   1. If not, change model structure or make model “larger”
   2. If you can overfit: Regularize
Toy Example: How to initialize W?

Let’s assume

\[ W = [-100, 100] \]

\[ X_j \in [-1, 1] \]

Our non linearity is \( \tanh \)  

If we init the weight to 100 the derivative would be? 0  

\[ \frac{d}{dx} \tanh(x) = 1 - \tan^2(x) \]

\[ Z = WX \]

Conclusion: If you initialize your weights badly and nothing will happen!
Toy Example: How to initialize W?

Intuitively, based on the statistics of X we want to fall in this area.
We initialized the biases to be 0 and the weights $W_{ij}$ at each layer with the following commonly used heuristic:

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right],$$

where $U[-a, a]$ is the uniform distribution in the interval $(-a, a)$ and $n$ is the size of the previous layer (the number of columns of $W$).
Parameter Initialization

In their empirical findings, [Glorot AISTATS 2010] suggest that for sigmoid and tanh activation units, lower error rates are achieved and faster convergence occurs when the weights of a matrix $W \in \mathbb{R}^{n_{(l+1)} \times n_{(l)}}$ are initialized randomly with a uniform distribution with the following range:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

with $n_i$ the size of layer $i$

Bias units are initialized to 0.

Guess how much is needed for sigmoid units? x4
General Strategy for Successful N Nets

1. Select network structure appropriate for problem
   1. Structure: Single words, fixed windows, bag of words, recursive vs. recurrent, CNN, sentence based vs. document
   2. Nonlinearity

2. Check for implementation bugs with gradient checks

3. Parameter initialization

4. Optimization tricks

5. Check if the model is powerful enough to overfit
   1. If not, change model structure or make model “larger”
   2. If you can overfit: Regularize
Stochastic Gradient Descent (SGD)

- Gradient descent uses total gradient over all examples per update, SGD updates after only 1 or few examples:

\[ \theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla_\theta J_t(\theta) \]

- \( J_t \) = loss function at current example, \( \theta \) = parameter vector, \( \alpha \) = learning rate.

- Ordinary gradient descent as a batch method is very slow, should never be used. Use \( 2^{\text{nd}} \) order batch method such as L–BFGS.

- On large datasets, SGD usually wins over all batch methods. On smaller datasets L-BFGS or Conjugate Gradients win. Large-batch L-BFGS extends the reach of L-BFGS [Le et al. ICML 2011].
Learning Rates

- Simplest recipe: keep it fixed and use the same for all parameters.
- Collobert scales them by the inverse of square root of the fan-in of each neuron.
- Better results can generally be obtained by allowing learning rates to decrease, typically in $O(1/t)$ because of theoretical convergence guarantees, e.g., with hyper-parameters $\epsilon_0$ and $\tau$:
  \[
  \epsilon_t = \frac{\epsilon_0 \tau}{\max(t, \tau)}
  \]
- Better yet: No hand-set learning rates by using L-BFGS or AdaGrad (Duchi, Hazan, & Singer 2011)
Always visualize you Learning Rates

With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line). This is because there is too much "energy" in the optimization and the parameters are bouncing around chaotically, unable to settle in a nice spot in the optimization landscape.
Adagrad

• Standard SGD, fixed alpha: \[ \theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta) \]

• Instead: Adaptive learning rates!

  Related paper: Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, Duchi et al. 2010

• Learning rate is adapting differently for each parameter and rare parameters get larger updates than frequently occurring parameters. Word vectors!

• Let gradient at time \( t \) for the \( i \)th element: \[ g_{t,i} = \frac{\partial}{\partial \theta_i} J_t(\theta) \]

  then \[ \theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i} \]

```
# Assume the gradient dx and parameter vector x

cache += dx**2

x += - learning_rate * dx / np.sqrt(cache + 1e-8)
```
General Strategy

1. Select appropriate Network Structure
   1. Structure: Single words, fixed windows vs Recursive Sentence Based vs Bag of words
   2. Nonlinearity
2. Check for implementation bugs with gradient check
3. Parameter initialization
4. Optimization tricks
5. Check if the model is powerful enough to overfit
   1. If not, change model structure or make model “larger”
   2. If you can overfit: Regularize

Assuming you found the right network structure, implemented it correctly, optimize it properly and you can make your model overfit on your training data.

Now, it’s time to regularize
Visualize the training vs validation error

The gap between the training and validation accuracy indicates the amount of overfitting.
Prevent Overfitting: Model Size and Regularization

- Simple first step: Reduce model size by lowering number of units and layers and other parameters

- Standard L1 or L2 regularization on weights. E.g:  
  \[ J_R = J + \lambda \sum_{i=1}^{L} \left\| W^{(i)} \right\|_F \]

- Early Training Stopping: Use parameters that gave best validation error

- Sparsity constraints on hidden activations, e.g., add to cost:  
  \[ KL \left( 1/N \sum_{n=1}^{N} a_i^{(n)} \right) \]
Regularization: Dropout
“randomly set some neurons to zero in the forward pass”

[Srivastava et al., 2014]
Example forward pass with a 3-layer network using dropout

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```
Waaaait a second...
How could this possibly be a good idea?

Forces the network to have a redundant representation.
Waaaait a second...
How could this possibly be a good idea?

Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.
At test time....

Ideally:
want to integrate out all the noise

Monte Carlo approximation:
do many forward passes with
different dropout masks, average all predictions
At test time....

Can in fact do this with a single forward pass! (approximately)
Leave all input neurons turned on (no dropout).

(this can be shown to be an approximation to evaluating the whole ensemble)
At test time....

Can in fact do this with a single forward pass! (approximately) Leave all input neurons turned on (no dropout).

Q: Suppose that with all inputs present at test time the output of this neuron is x.

What would its output be during training time, in expectation? (e.g. if p = 0.5)
At test time....
Can in fact do this with a single forward pass! (approximately)
Leave all input neurons turned on (no dropout).

during test: \( a = w_0 x + w_1 y \)

during train:
\[
E[a] = \frac{1}{4} * (w_0 * 0 + w_1 * 0 \\
        + w_0 * 0 + w_1 * y \\
        + w_0 * x + w_1 * 0 \\
        + w_0 * x + w_1 * y)
\]
\[
= \frac{1}{4} * (2 w_0 x + 2 w_1 y) \\
= \frac{1}{2} * (w_0 x + w_1 y)
\]
At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

during test:  \( a = w_0 x + w_1 y \)

during train:

\[
E[a] = \frac{1}{4} \left( w_0^2 + w_1^2 + w_0 x + w_1 y \right)
\]

With \( p=0.5 \), using all inputs in the forward pass would inflate the activations by 2x from what the network was "used to" during training!

=> Have to compensate by scaling the activations back down by \( \frac{1}{2} \)

\[
= \frac{1}{4} \left( 2 w_0 x + 2 w_1 y \right) \\
= \frac{1}{2} \left( w_0 x + w_1 y \right)
\]
We can do something approximate analytically

```python
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
   output at test time = expected output at training time
Dropout Summary

**Vanilla Dropout**: Not recommended implementation (see notes below)

```python
p = 0.5  # probability of keeping a unit active. higher = less dropout

def train_step(X):
    
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p  # first dropout mask
    H1 *= U1  # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p  # second dropout mask
    H2 *= U2  # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

- **Drop in forward pass**: 
  - `U1 = np.random.rand(*H1.shape) < p`  
  - `H1 *= U1`  

- **Scale at test time**: 
  - `H1 = np.maximum(0, np.dot(W1, X) + b1) * p`  
  - `H2 = np.maximum(0, np.dot(W2, H1) + b2) * p`  

More common: “Inverted dropout”

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```
test time is unchanged!
Deep Learning Tricks of the Trade

  - Unsupervised pre-training
  - Stochastic gradient descent and setting learning rates
  - Main hyper-parameters
    - Learning rate schedule & early stopping
    - Minibatches
    - Parameter initialization
    - Number of hidden units
    - L1 or L2 weight decay
    - Sparsity regularization

- How to efficiently search for hyper-parameter configurations
  - Short answer: Random hyperparameter search (!)
Summary

- Gradient check your implementation with a small batch of data and be aware of the pitfalls.
- As a sanity check, make sure your initial loss is reasonable, and that you can achieve 100% training accuracy on a very small portion of the data.
- During training, monitor the loss, the training/validation accuracy, and if you're feeling fancier, the magnitude of updates in relation to parameter values (it should be ~1e-3), and when dealing with ConvNets, the first-layer weights.
- The two recommended updates to use are either SGD+Nesterov Momentum or Adam.
- Decay your learning rate over the period of the training. For example, halve the learning rate after a fixed number of epochs, or whenever the validation accuracy tops off.
- Search for good hyperparameters with random search (not grid search). Stage your search from coarse (wide hyperparameter ranges, training only for 1-5 epochs), to fine (narrower rangers, training for many more epochs).
- Form model ensembles for extra performance.
Advanced Tricks

• Multi Task Learning
• Adversarial Examples
Multi-task learning / Weight sharing

- Similar to neural network from last class but replaces the single scalar score with a \textit{Softmax} classifier

- Training is again done via backpropagation which gives an error similar to the score in the scoring learning model

- NLP (almost) from scratch, Collobert et al. 2011

\[
\hat{y} = \text{softmax} \left( W^{(S)} f(Wx + b) \right)
\]
The Model – Training

- We already know the softmax classifier and how to optimize it.
- The interesting twist in deep learning is that the input features $x$ are also learned, similar to learning with a score:

$$s = x^T a_1 W_{23} a_2 U_2$$
The Model – Training

- **Main additional idea:** We can share both the word vectors AND the hidden layer weights. Only the softmax weights are different.

- **Cost function** is just the sum of two cross entropy errors

\[
\hat{y}^{(1)} = \text{softmax} \left( W^{(S_1)} f(W x + b) \right) \quad \hat{y}^{(2)} = \text{softmax} \left( W^{(S_2)} f(W x + b) \right)
\]
The Model – Training

- **Example**: predict each window’s center NER tag and POS tag (e.g., DT, NN, NNP, JJ, JJS (superlatives adj), VB, ...)

- **Efficient implementation**: same forward prop, compute errors on hidden vectors and add $\delta^{total} = \delta^{NER} + \delta^{POS}$
The secret sauce is the unsupervised word vector pre-training on a large text collection

<table>
<thead>
<tr>
<th></th>
<th>POS WSJ (acc.)</th>
<th>NER CoNLL (F1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-of-the-art*</td>
<td>97.24</td>
<td>89.31</td>
</tr>
<tr>
<td>Supervised NN</td>
<td>96.37</td>
<td>81.47</td>
</tr>
<tr>
<td>Word vector pre---training</td>
<td>97.20</td>
<td>88.87</td>
</tr>
<tr>
<td>followed by supervised NN**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ hand---crafted features***</td>
<td>97.29</td>
<td>89.59</td>
</tr>
</tbody>
</table>

* Representative systems: POS: (Toutanova et al. 2003), NER: (Ando & Zhang 2005)

** 130,000---word embedding trained on Wikipedia and Reuters with 11 word window, 100 unit hidden layer – then supervised task training

***Features are character suffixes for POS and a gazetteer for NER
Supervised refinement of the unsupervised word representation helps

<table>
<thead>
<tr>
<th></th>
<th>POS WSJ (acc.)</th>
<th>NER CoNLL (F1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervised NN</td>
<td>96.37</td>
<td>81.47</td>
</tr>
<tr>
<td>NN with Brown clusters</td>
<td>96.92</td>
<td>87.15</td>
</tr>
<tr>
<td>Fixed embeddings*</td>
<td>97.10</td>
<td>88.87</td>
</tr>
<tr>
<td><strong>C&amp;W 2011</strong></td>
<td>97.29</td>
<td>89.59</td>
</tr>
</tbody>
</table>

* Same architecture as C&W 2011, but word embeddings are kept constant during the supervised training phase

** C&W is unsupervised pre-train + supervised NN + features model of last slide
Adversarial Examples

- **What is adversarial example?** We can let the network to misclassify an image by adding a imperceptible (for human) perturbation.

- **Why do adversarial examples exist?** Deep Neural Networks learn input-output mappings that are discontinuous to a significant extent.

- **Interesting observation:** the adversarial examples generated for network A can also make network B fail.
Let's fool a binary linear classifier

\[ P(y = 1 \mid x; w, b) = \frac{1}{1 + e^{-(w^T x + b)}} = \sigma(w^T x + b) \]

Since the probabilities of class 1 and 0 sum to one, the probability for class 0 is
\[ P(y = 0 \mid x; w, b) = 1 - P(y = 1 \mid x; w, b) \]
Hence, an example is classified as a positive example \((y = 1)\) if
\[ \sigma(w^T x + b) > 0.5, \]
or equivalently if the score \(w^T x + b > 0\).
Let's fool a binary linear classifier

class 1 score = dot product:
= -2 + 1 + 3 + 2 + 2 - 2 + 1 - 4 - 5 + 1 = -3
=> probability of class 1 is $\frac{1}{1+e^{(-(-3))}} = 0.0474$
i.e. the classifier is 95% certain that this is class 0 example.
class 1 score = dot product:
= -2 + 1 + 3 + 2 + 2 - 2 + 1 - 4 - 5 + 1 = -3
=> probability of class 1 is $1/(1+e^{-(3)}) = 0.0474$
i.e. the classifier is 95% certain that this is class 0 example.

-1.5+1.5+3.5+2.5+2.5-1.5+1.5-3.5-4.5+1.5 = 2
=> probability of class 1 is now $1/(1+e^{-(2)}) = 0.88$
i.e. we improved the class 1 probability from 5% to 88%
mix in a tiny bit of Goldfish classifier weights

0.9% bobsled + 100.0% goldfish = 100% Goldfish
Generate Adversarial Examples

Input image: \( x \in \mathbb{R}^m \)

Classifier: \( f : \mathbb{R}^m \rightarrow \{1 \ldots k\} \)

Target label: \( l \in \{1 \ldots k\} \)

Minimize \( \|r\|_2 \) subject to:
1. \( f(x + r) = l \)
2. \( x + r \in [0, 1]^m \)

When \( f(x) \neq l \):

Minimize \( c|r| + \text{loss}_f(x + r, l) \) subject to \( x + r \in [0, 1]^m \)

\( x+r \) is the closest image to \( x \) classified as \( l \) by \( f \).
Intriguing properties

• Properties:
  • Visually hard to distinguish the generated adversarial examples.
  • Cross model generalization. (different hyper-parameters)
  • Cross training-set generalization. (different training set)

• Observation:
  • adversarial examples are universal.
  • back-feeding adversarial examples to training might improve generalization of the model.