Midterm Study Guide

My Guide Yay

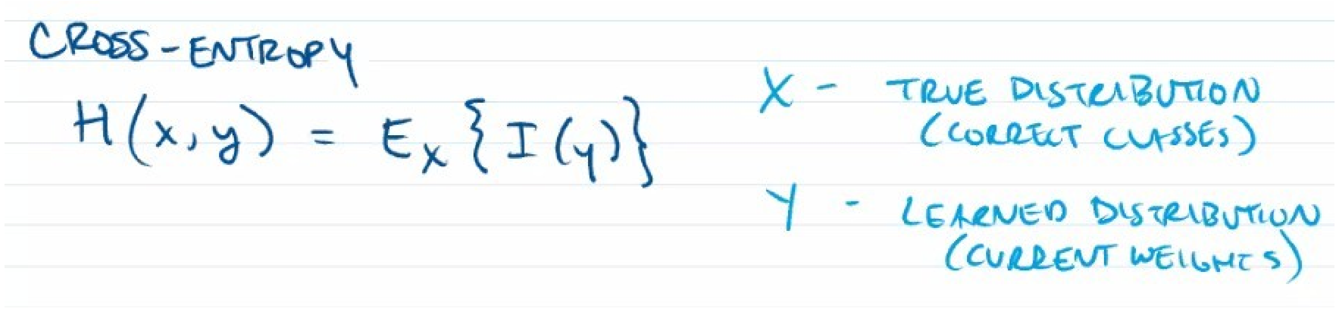
Chapter 1 – An Introduction to Neural Networks

General Unsorted Stuff

Stochastic Gradient Descent

Stochastic Gradient Descent is gradient descent where we randomize the order of the data before each epoch.

Softmax and Cross-Entropy Loss



CROSS-ENTROPY

$$H(x, y) = E_x\{I(y)\}$$

X - TRUE DISTRIBUTION (CORRECT CLASSES)

Y - LEARNED DISTRIBUTION (CURRENT WEIGHTS)

> Put another way, the cross-entropy is the difference in information between different distributions. We use this as the loss, because we're trying to minimize the difference, because we're trying to make the learned distributions (via weights) match the true distribution.

So this is why we use cross-entropy loss with softmax. We're calculating the ***entropy*** aka the difference in information between the base truth and the output. And trying to minimize the ***entropy***.

1.2 – The Basic Architecture of Neural Networks

1.2.1 Single Computational Layer: The Perceptron

Basically we just take the sign of the bias plus the weighted sum, so {-1, 1}. Positive sign means "yes", negative means "no". Can only solve linearly separable problems.

Perceptron Criterion

The loss function is a staircase, which can't be run through gradient descent because it's not continuous. We can use a version called the ***perceptron criterion***:

$$\nabla L_{\text{smooth}} = \sum_{(\overline{X},y)\in\mathcal{D}} (y-\hat{y})\overline{X}$$

Perceptron Update Rule

For a normal Perceptron, we update ***all*** weights whenever we do an update. However, we only do an update when the prediction is incorrect.

$$\overline{W} \Leftarrow \overline{W} + \alpha(y-\hat{y})\overline{X}$$

1.2.1.1 – What Objective Function Is the Perceptron Optimizing?

> The modified loss function to enable gradient computation of a non-differentiable function is also referred to as a smoothed surrogate loss function.

1.2.1.5 – Choice of Loss Function

Loss functions pair with gradient descent functions. Or something.

Target Type	**Activation**	**Loss Function**
Binary targets (Logistic Regression)	Identity	L = log(1 + exp(−y · ŷ))
Categorical targets	Softmax	L = −log(ŷ r)

1.2.2 – Multilayer Neural Networks

Activation functions and their derivatives; Derivatives are in terms of the original function's output; v is the pre-activation value.

Name	**Output**	**Derivative**
Identify	$O=v$	$\frac{\partial o}{\partial v}=1$
Sigmoid	$o=\frac{1}{1+\exp(-v)}$	$\frac{\partial o}{\partial v}=o(1-o)$
Tanh	$o=\frac{\exp(2v)-1}{\exp(2v)+1}$	$\frac{\partial o}{\partial v}=1-o^2$

1.3 – Training a Neural Network with Backpropagation

1.4 – Practical Issues in Neural Network training

1.4.1 – The Problem of Overfitting

Yep. Try to avoid over-fitting via various techniques.

1.4.1.1 – Regularization

We would typically include a penalty term to the *Loss Function* of something like $\lambda||\overline{W}||^p$, where ***p==2*** so it's usually ***Tikhonov regularization*** $\lambda||\overline{W}||^2$

We could take the perceptron update rule $\overline{W} \Leftarrow \overline{W} + \alpha \sum_{\overline{X} \in S} E(\overline{X})\overline{X}$ and modify it to include regularization like so:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{\overline{X} \in S} E(\overline{X})\overline{X}$$

Where the E function refers to the error, or difference between ***y*** and ***y-hat***. This essentially causes "weight decay" of unimportant features.

1.4.1.2 – Neural Architecture and Parameter Sharing

Basically try to design the net with some thought for how the data actually is. Examples:

- Images use ***convolutional*** nets (connect nearby pixels to each other on the next layer but not ALL pixels to each other).
- ***Recurrent*** nets in order to handle data where one observation is temporally related to the previous or next

1.4.1.3 – Early Stopping

Basically stop training once we hit a certain threshold, such as a decent loss, or a number of iterations.

1.4.1.4 – Trading Off Breadth for Depth

Make the network deeper instead of wider layers. This can improve computational time as well as reduce over-fitting.

1.4.2 - The Vanishing and Exploding Gradient Problems

Basically when a network is very deep, we end up multiplying many times to perform gradient descent (backprop), which means the final value could approach zero and become truncated due to the limitations of IEEE floating point storage. Exploding gradient is the same problem: Overly large values that happen with deep networks.

1.4.4 – Local and Spurious Optima

Basically we can get in local minima, and spurious optima (stuck in local minima that sucks). We can try to avoid this by pre-training the weights; Calculating initial weight values with some sort of algorithm, or even training parts of the network without the whole network, first.

1.5 – The Secrets to the Power of Function Composition

1.5.1 – The Importance of Nonlinear Activation

Most important tidbit I don't know: Having multiple layers allows earlier layers to perform transformation of the data, such that the data eventually becomes linearly separable by the time we get to the last layer.

> In other words, the activation functions enable nonlinear transformations of the data, that become increasingly powerful with multiple layers.

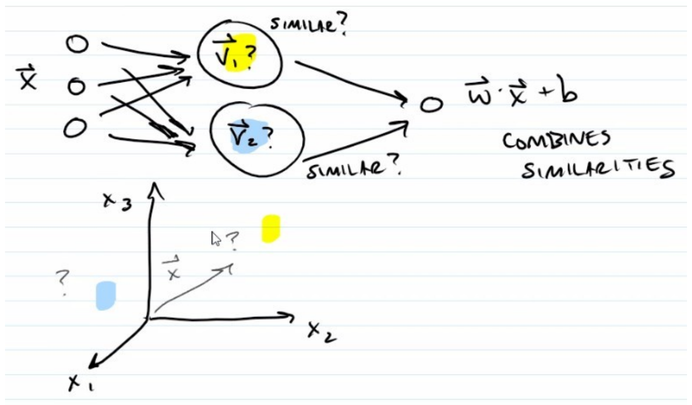
1.5.2 – Reducing Parameter Requirements with Depth

Basically re-iterates that we can reduce the number of parameters exponentially by making the network more deep, and that we're able to do this because the layers organize themselves into hierarchies of ideas. But also that deep networks still have their own problems, like vanishing/exploding gradients or instability with respect to parameter selection.

1.6 – Common Neural Architectures

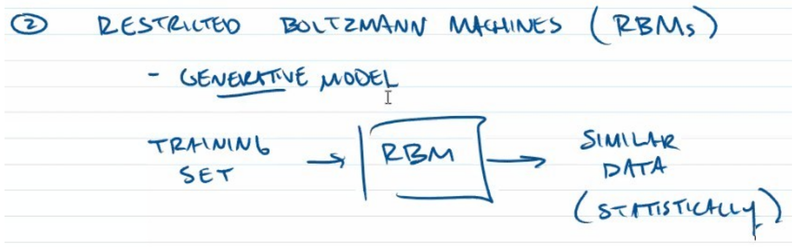
1.6.2 – Radial Basis Function Networks

Typically only two layers, and they will activate based on their similarity to each other.



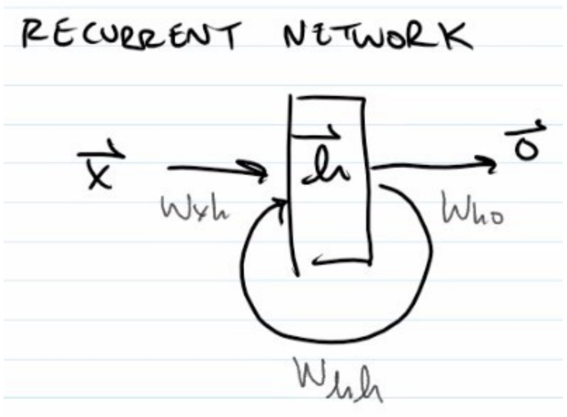
1.6.3 – Restricted Boltzmann Machines (RBMs)

An RMB is what's called a generative model. The idea is you'll give it some input as the training set, you'll feed it into the RMB. Then, the output it produces is going to be statistically similar data (e.g., with the same distribution).


② RESTRICTED BOLTZMANN MACHINES (RBMs)
- GENERATIVE MODEL
TRAINING SET
RBM
SIMILAR DATA (STATISTICALLY)

1.6.4 – Recurrent Neural Networks

Basically looping some outputs back to hidden layers, or hiddens back to hiddens, in order to create quasi short term memory ability. Very useful for time-series or sequenced data.



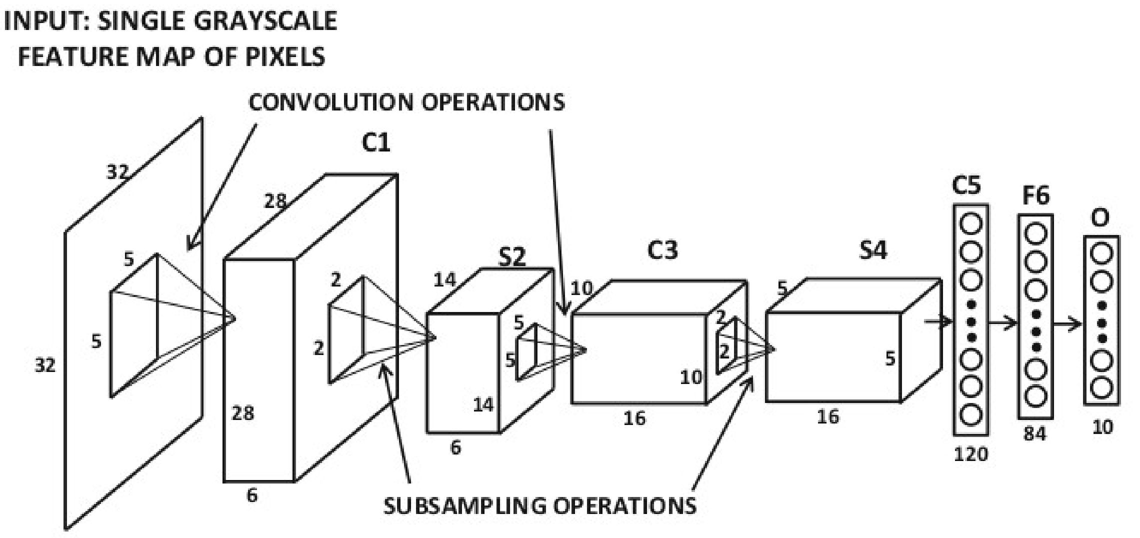
1.6.5 – Convolutional Neural Networks

Earlier layers are sparsely connected, perhaps each neuron is connected to only a few clustered neuron in the layer before. Fully connected layers appear later in the network.

Just with feed-forward networks, the premise is hierarchy based on layer number. But in convolutional, earlier layers grab simple shapes, and more complex shapes are identified in later layers. Eventually, fully connected layers make decisions based on what the sparse layers determined.

Subsampling operations seem to merely be "averaging a group of pixels down to one pixel".

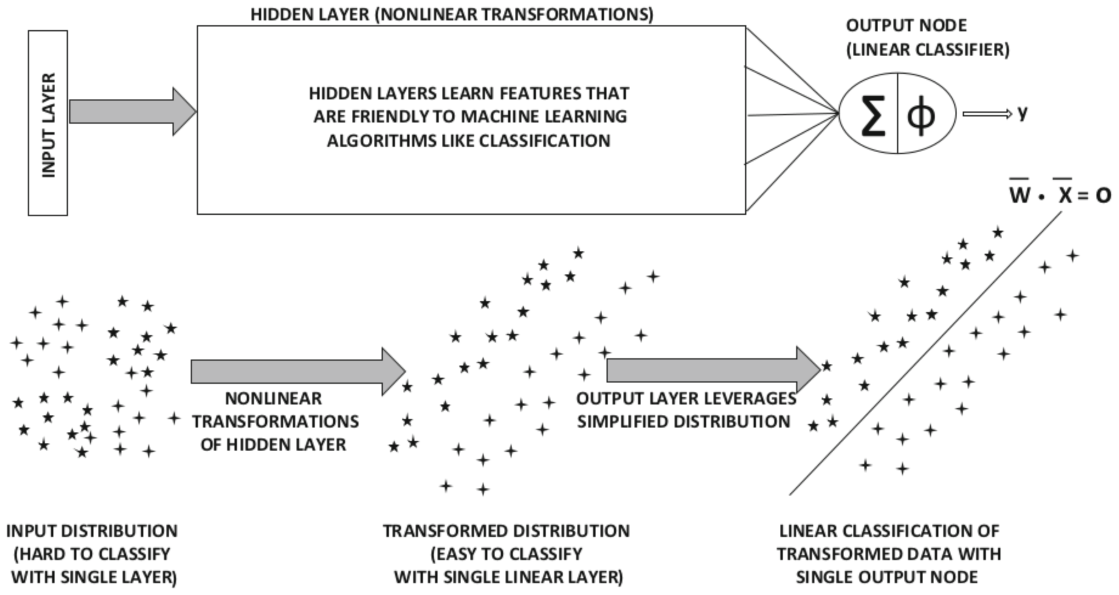
Inspired by biological eyeball/neurons in cats.



1.6.6 – Hierarchical Feature Engineering and Pretrained Models

The basic idea here is that each layer successively applies transformations to non-linear input data (or data from the previous layer), until eventually, the final output layer only has to make linear decisions.

Also, sometimes we may use ***pre-trained models***, where we basically take a model and train it to some simpler task. Then, once the model successfully learns the task, we may connect its output layer to a NEW set of layers in the hopes of learning an ***<u>even more complex</u>*** problem. In this case, the ***pre-trained*** portion of the model would have fixed weights, so we only need to worry about the newly added layers. Kinda cool.


HIDDEN LAYER (NONLINEAR TRANSFORMATIONS)
INPUT LAYER
HIDDEN LAYERS LEARN FEATURES THAT ARE FRIENDLY TO MACHINE LEARNING ALGORITHMS LIKE CLASSIFICATION
OUTPUT NODE (LINEAR CLASSIFIER)
Σ | Φ
y
W̄ • X̄ = O
NONLINEAR TRANSFORMATIONS OF HIDDEN LAYER
OUTPUT LAYER LEVERAGES SIMPLIFIED DISTRIBUTION
INPUT DISTRIBUTION (HARD TO CLASSIFY WITH SINGLE LAYER)
TRANSFORMED DISTRIBUTION (EASY TO CLASSIFY WITH SINGLE LINEAR LAYER)
LINEAR CLASSIFICATION OF TRANSFORMED DATA WITH SINGLE OUTPUT NODE

1.7 – Advanced Topics

1.7.1 – Reinforcement Learning

This is basically "global search", or basically training without targets, like with genetic algorithms.

1.7.2 – Separating Data Storage and Computations

> Neural networks that can tightly control access in reading and writing to an external memory are referred to as neural Turing machines [158] or memory networks [528]. Although these methods are advanced variants of recurrent neural networks, they show significantly improved potential than their predecessors.

1.7.3 – Generative Adversarial Networks

Two players:

- Generator → Tries to generate fake data or observations
- Discriminator → Tries to tell the difference between the Generator's fake data, and the real data

The result is the Generator eventually learns to make really really convincing fake data, which can then be used to train the real neural network.

1.8 – Two Notable Benchmarks

1.8.1 – The MNIST Database of Handwritten Digits

Collection of 60,000 training images and 10,000 testing images. Each image is just a digit, centered, and in black and white.

1.8.2 – The ImageNet Database

Huge database of 14 million images from over 1,000 categories. Also organized based on something called the WorldNet Hierarchy of Nouns.

Very popular for training image based networks. They run yearly contests.

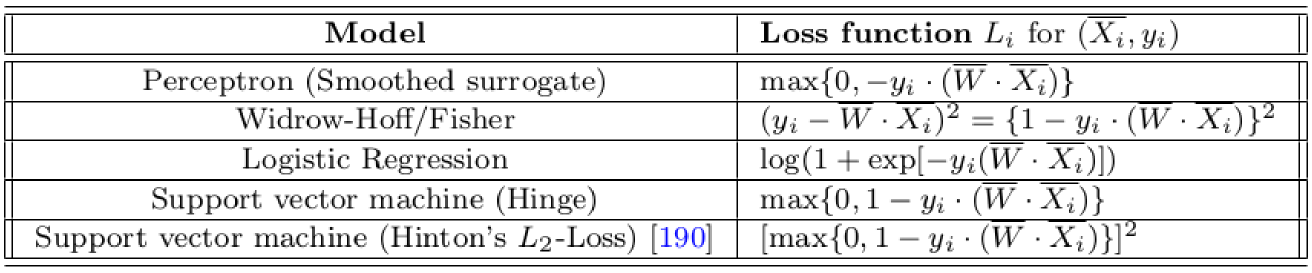
Chapter 2 – Machine Learning with Shallow Neural Networks

2.1 – Introduction

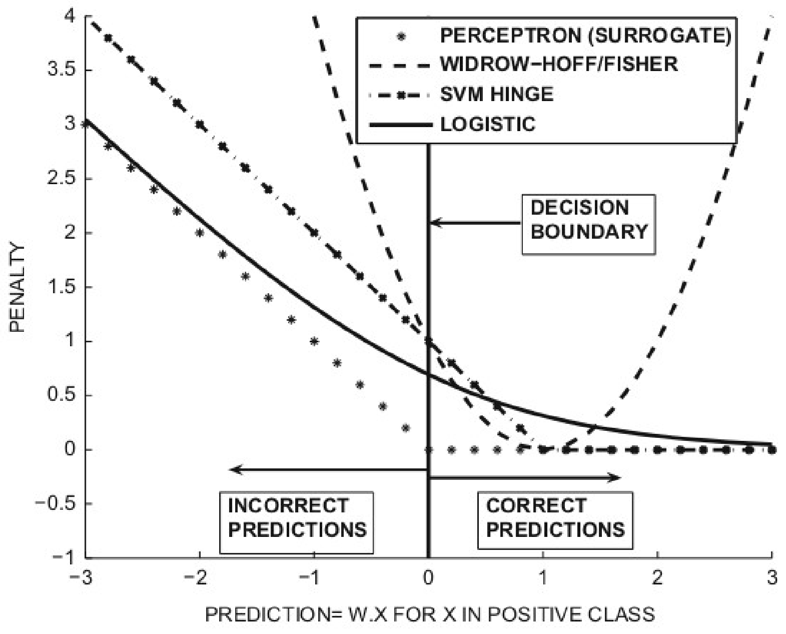
Basically shallow networks can be tweaked to represent traditional machine learning algorithms. But neural networks can have more complex representation (e.g., topology, loss functions, etc) that more closely fit the problem at hand, plus they can solve hierarchically, thus are much more powerful (although sometimes are overkill).

2.2 – Neural Architectures for Binary Classification Models

Uhm. Here's a quick summary of losses for various classifiers, found nearer the end of this subsection:



Model	**Loss function** L_i **for** $(\overline{X_i}, y_i)$
Perceptron (Smoothed surrogate)	$\max\{0, -y_i \cdot (\overline{W} \cdot \overline{X_i})\}$
Widrow-Hoff/Fisher	$(y_i - \overline{W} \cdot \overline{X_i})^2 = \{1 - y_i \cdot (\overline{W} \cdot \overline{X_i})\}^2$
Logistic Regression	$\log(1 + \exp[-y_i(\overline{W} \cdot \overline{X_i})])$
Support vector machine (Hinge)	$\max\{0, 1 - y_i \cdot (\overline{W} \cdot \overline{X_i})\}$
Support vector machine (Hinton's L_2-Loss) [190]	$[\max\{0, 1 - y_i \cdot (\overline{W} \cdot \overline{X_i})\}]^2$



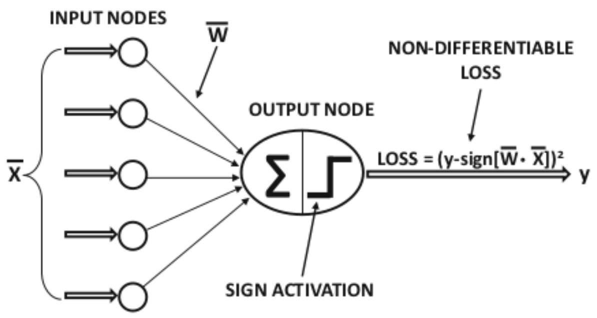
2.2.1 – Revisiting the Perceptron

Remember the Perceptron equation: $\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha(y_i - \hat{y}_i)\overline{X_i}$

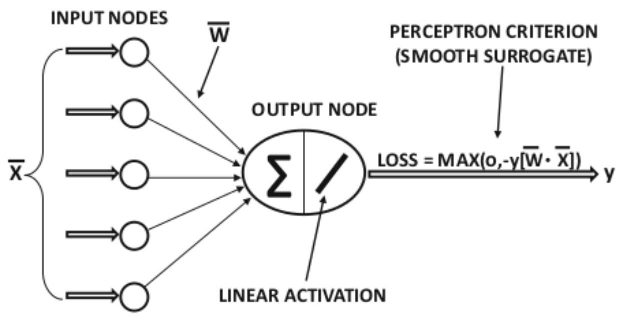
Also remember it includes a regularization term (left part $\bar{W}(1-\alpha\lambda)$), in addition to the actual update part $\alpha(y_i-\hat{y}_i)\bar{X}_i$.

> ***Indicator function:*** may be written using the indicator function I(·) ∈ {0, 1} that takes on 1 when the condition in its argument is satisfied

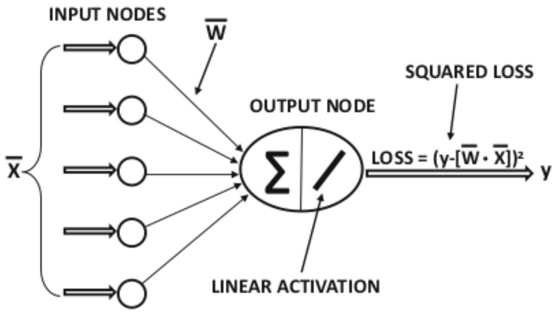
Probably some important variations to remember, lol:



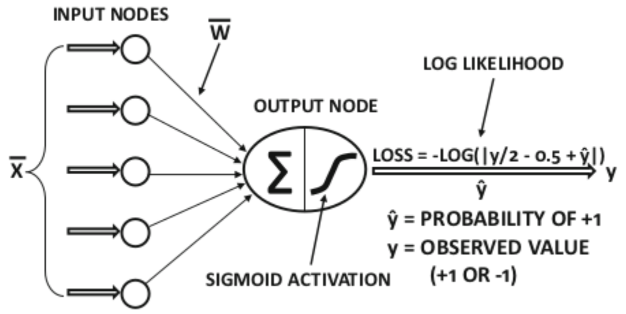
(a) The perceptron (discrete output)


INPUT NODES
$\overline{W}$
PERCEPTRON CRITERION (SMOOTH SURROGATE)
OUTPUT NODE
$\overline{X}$
LOSS = MAX(0,-y[$\overline{W}\cdot\overline{X}$])
y
LINEAR ACTIVATION

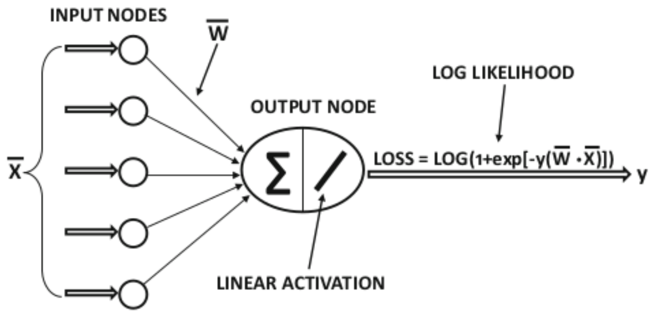
(b) The perceptron (continuous output)



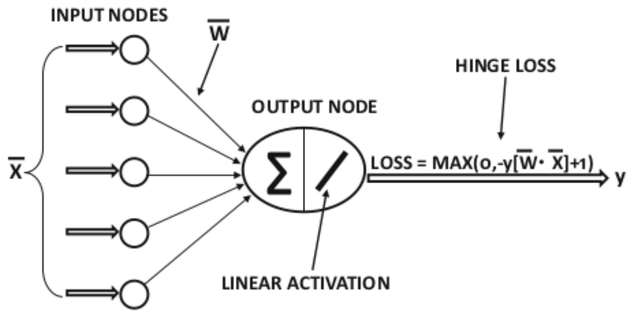
(c) Linear regression



(d) Logistic regression



(e) Logistic regression (alternate)



(f) Support vector machine

2.2.2 – Least-Squares Regression

If we calculate the error, and then square it, we end up using least-squares and we can just take the partial derivative of the squared error with respect to the weights:

This gradient can be computed as follows:

$$\frac{\partial e_i^2}{\partial \overline{W}} = -e_i \overline{X_i}$$

Also recall that the gradient here is negative (for some reason lol), and we’re using gradient ***<u>descent</u>***, which means go in the opposite direction of the gradient, so we can decrease it. That’s why we end up doing W = W + alpha(e) instead of W = W – alpha(e), essentially.

2.2.2.1 – Widrow-Hoff Learning

This is basically the perceptron, but with identity activation instead of sign activation. It has the bad effect of penalizing “too correct” predictions.

2.2.3 - Logistic Regression

Take the output value and run it through sigmoid, so we get a likelihood for a class, rather than applying sign activation. This also means we use the *negative log loss*.

$$\hat{y}_i = P(y_i = 1) = \frac{1}{1 + \exp(-\overline{W} \cdot \overline{X_i})}$$

2.3 – Neural Architectures for Multiclass Models

2.3.1 – Multiclass Perceptron

Uhm. I think we’re doing one set of weights for each class here, and only taking the strongest output as the class prediction.

The update rule only gets applied when there is a misclassification, and goes as follows:

$$\overline{W_r} \Leftarrow \overline{W_r} + \begin{cases} \alpha \overline{X_i} & \text{if } r = c(i) \\ -\alpha \overline{X_i} & \text{if } r \neq c(i) \text{ is most misclassified prediction} \\ 0 & \text{otherwise} \end{cases} \qquad (2.24)$$

2.3.2 – Weston-Watkins SVM

> the Weston-Watkins SVM updates the separator of any class that is predicted more favorably than the true class.
>
> Not only does the Weston-Watkins SVM update the separator in the case of misclassification, it updates the separators in cases where an incorrect class

> gets a prediction that is "uncomfortably close" to the true class. This is based on the notion of margin.

2.3.3 – Multinomial Logistic Regression (Softmax Classifier)

Multinomial logistic regression uses softmax on its outputs:

$$P(r|\overline{X_i}) = \frac{\exp(\overline{W_r} \cdot \overline{X_i})}{\sum_{j=1}^{k} \exp(\overline{W_j} \cdot \overline{X_i})}$$

It also then uses negative log-likelihood loss.

$$\begin{aligned} L_i &= -\log[P(c(i)|\overline{X_i})] \\ &= -\overline{W}_{c(i)} \cdot \overline{X_i} + \log[\sum_{j=1}^{k} \exp(\overline{W_j} \cdot \overline{X_i})] \\ &= -v_{c(i)} + \log[\sum_{j=1}^{k} \exp(v_j)] \end{aligned}$$

$$\overline{W_r} \Leftarrow \overline{W_r}(1 - \alpha\lambda) + \alpha \begin{cases} \overline{X_i} \cdot (1 - P(r|\overline{X_i})) & \text{if } r = c(i) \\ -\overline{X_i} \cdot P(r|\overline{X_i}) & \text{if } r \neq c(i) \end{cases}$$

> The softmax classifier updates all the k separators for each training instance, unlike the multiclass perceptron and the Weston-Watkins SVM, each of which updates only a small subset of separators (or no separator) for each training instance.

I think it's saying that UHHHMMMM THOMAS HELP

Chapter 3 – Training Deep Neural Networks

3.2 – Backpropagation – The Gory Details

3.2.6 – A Decoupled View of Vector-Centric Backpropagation

To aid understanding, we could consider the weighted sum as a different layer from the activation:

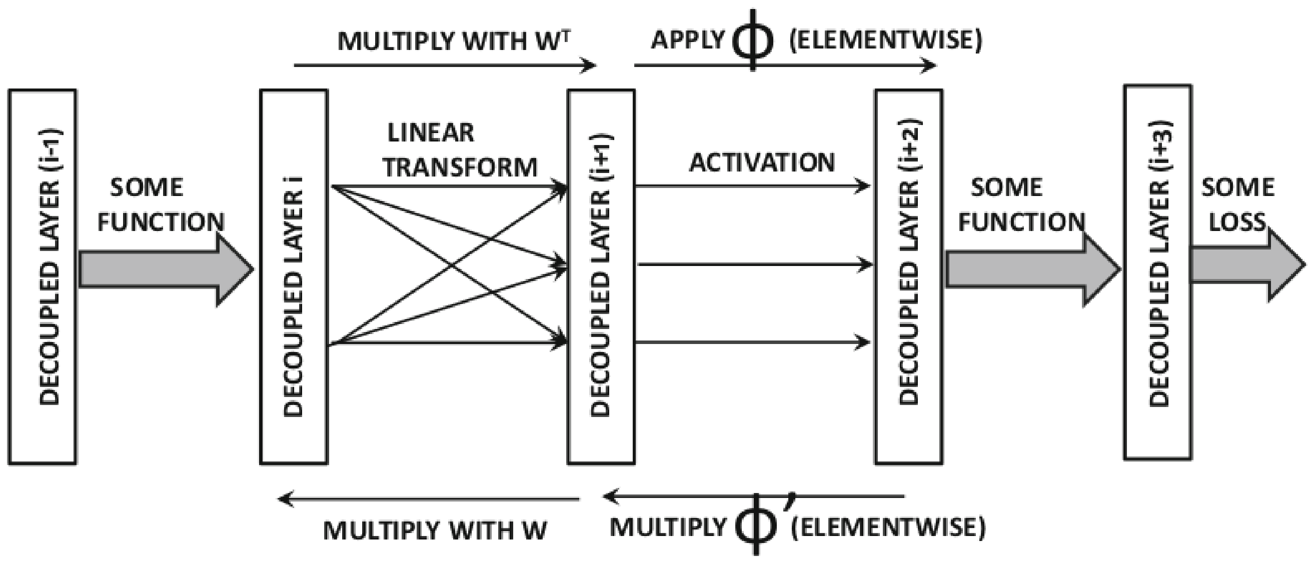
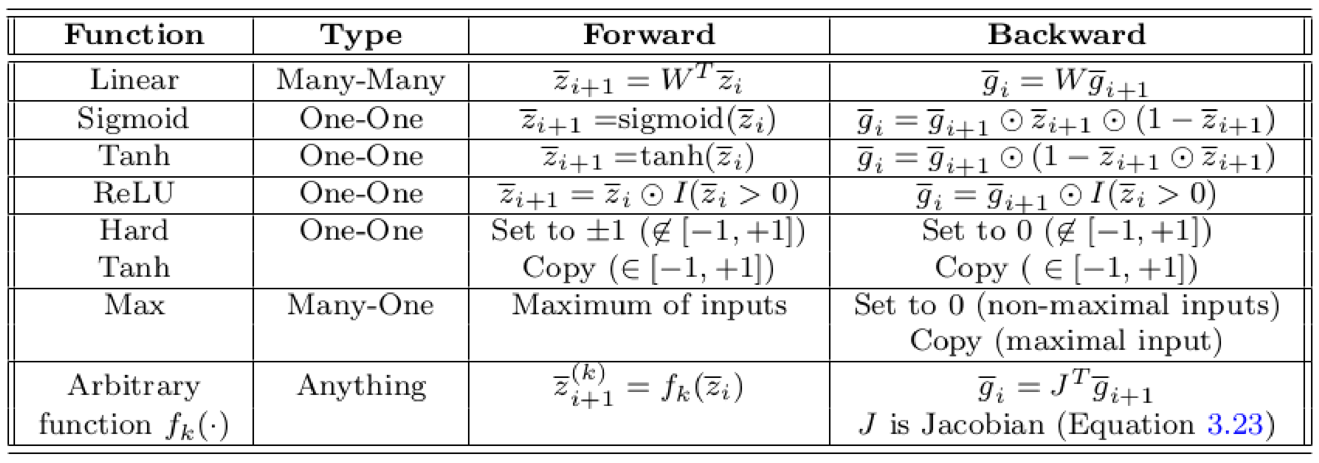


Figure 3.7: A decoupled view of backpropagation

With this in mind, we could use the following table to remember how to calculate during forward and backward propagation:

> Table 3.1: Examples of different functions and their backpropagation updates between layers i and (i + 1). The hidden values and gradients in layer i are denoted by z i and g i . Some of these computations use I(·) as the binary indicator function.



Function	Type	Forward	Backward
Linear	Many-Many	$\overline{z}_{i+1} = W^T \overline{z}_i$	$\overline{g}_i = W \overline{g}_{i+1}$
Sigmoid	One-One	$\overline{z}_{i+1} =$sigmoid$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (1 - \overline{z}_{i+1})$
Tanh	One-One	$\overline{z}_{i+1} =$tanh$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot (1 - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	One-One	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	One-One	Set to ± 1 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Many-One	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)
Arbitrary function $f_k(\cdot)$	Anything	$\overline{z}_{i+1}^{(k)} = f_k(\overline{z}_i)$	$\overline{g}_i = J^T \overline{g}_{i+1}$ J is Jacobian (Equation 3.23)

> […] Therefore, the backward propagation operation is just like forward propagation. Given the vector of gradients in a layer, one only has to apply the

> operations shown in the final column of Table 3.1 to obtain the gradients with respect to the previous layer.

3.2.8 – Mini-Batch Stochastic Gradient Descent

It seems like they're saying purely stochastic gradient descent is a big unstable or imprecise, because we ***wish*** we could compute the gradient for the ***entire dataset*** at the same time, before making an update. But we're limited by memory and such. So instead, we randomly select several observations (the ***mini batch****)*, compute all their gradients, then adjust the network at once using the *sum of the gradients*.

This is supposed to be a trade-off between stability+accuracy against RAM requirements and such. And apparently, the batch size doesn't need to be very large to see significant benefits; We wouldn't see much benefit after a certain point, meaning *mini-batch stochastic gradient descent* is virtually as good as doing the whole training set at once.

3.2.10 – Checking the Correctness of Gradient Computation

We can basically check our backprop algorithm by making a slight change to one weight, then checking that the difference in loss changes at the rate rate as the gradient we calculated (with respect to that one weight). We could then also do this check at a few random spots every so often, so enjoy a sort-of built-in sanity check on our algorithm.

> The weight of this edge is perturbed by adding a small amount # > 0 to it. Then, the forward algorithm is executed with this perturbed weight and the loss L(w + #) is computed. Then, the partial derivative of the loss with respect to w can be shown to be the following:

$$\frac{\partial L(w)}{\partial w} \approx \frac{L(w+\epsilon) - L(w)}{\epsilon}$$

> When the partial derivatives do not match closely enough, it is easy to detect that an error must have occurred in computation. One needs to perform the above estimation for only two or three checkpoints in the training process, which is quite efficient. However, it might be advisable to perform the checking over a large subset of the parameters at these checkpoints.
>
> However, it might be advisable to perform the checking over a large subset of the parameters at these checkpoints. One problem is in determining when the

gradients are “close enough,” especially when one has no idea about the absolute magnitudes of these values. This is achieved by using relative ratios.

Let the backpropagation-determined derivative be denoted by G e , and the aforementioned estimation be denoted by G a . Then, the relative ratio ρ is defined as follows:

$$\rho = \frac{|G_e - G_a|}{|G_e + G_a|}$$

Typically, the ratio should be less than 10 −6 , although for some activation functions like the ReLU in which sharp changes in derivatives occur at particular points, it is possible for the numerical gradient to be different from the computed gradient. In such cases, the ratio should still be less than 10 −3 .